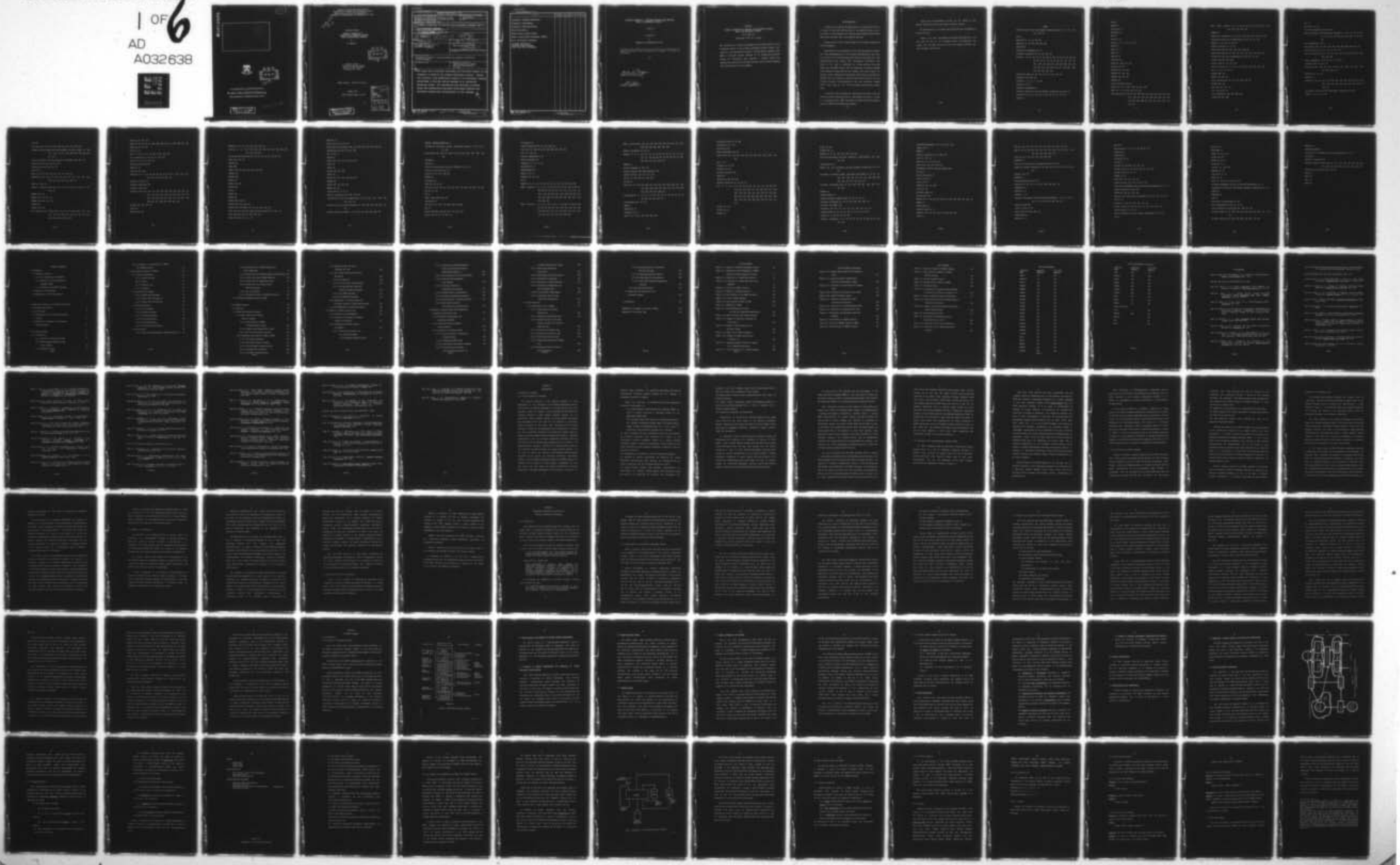
AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416
UNCLASSIFIED 76-05 NL

1 OF 6
AD
A032638



12

UNIVERSITY of PENNSYLVANIA

The Moore School of Electrical Engineering

PHILADELPHIA, PENNSYLVANIA 19174

Automatic Program Generation Project
Department of Computer and Information Science
Moore School of Electrical Engineering
UNIVERSITY OF PENNSYLVANIA, Philadelphia, Pa. 19174

Code 23
O.S.

Technical Report

AUTOMATIC GENERATION OF BUSINESS DATA PROCESSING PROGRAMS FROM A NON-PROCEDURAL LANGUAGE

by

N. Adam Rin

Prepare for The
Information Systems Program
Office of Naval Research
Arlington, Va. 22217

Under Contract N00014-76-C-0416

October 1976

Moore School Report # 76-05

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Department of Computer and Information Science The Moore School of Electrical Engineering (D2) University of Pennsylvania Philadelphia, Pennsylvania 19174	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

(6) Automatic Generation of Business Data-Processing Programs from a Non-Procedural Language.

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

(9) Technical Report, - October 1976

5. AUTHOR(S) (First name, middle initial, last name)

N. Adam Rin (10) (11) 26 oct 76 (12) 489 p.

6. REPORT DATE October 20, 1976	7a. TOTAL NO. OF PAGES 440	7b. NO. OF REFS 65

8a. CONTRACT OR GRANT NO. (15) N00014-76-C-0416 b. PROJECT NO. NR 049-153 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Moore School Report 76-05 (14) 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

Reproduction in whole or in part permitted for purposes of the United States Government.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Information Systems Program Office of Naval Research U.S. Navy

13. ABSTRACT

The rising cost of software development has motivated research on the automation of parts of the software development process. Towards that objective, this dissertation presents a non-procedural language called MODEL to describe desired programs of an information processing system. The dissertation also describes a software system that automatically generates conventional business data processing programs from specifications in that language.

237 000

Unclassified

Security Classification

Unclassified

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
automatic program generation						
automatic programming						
automatic code generation						
data processing						
graph theory applications						
Module Description Language (MODEL)						
non- procedural language						
program generation						
program specification						
very high level language						

Unclassified

Security Classification

AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM A NON-PROCEDURAL LANGUAGE

N. Adam Rin

A DISSERTATION

in

Computer and Information Science

Presented to the Graduate Faculty of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1976

Noah S. Prywes

Supervisor of Dissertation

A.K. Joshi

Graduate Group Chairman

ABSTRACT

AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM A NON-PROCEDURAL LANGUAGE

by N. Adam Rin

Supervisor: Prof. N. S. Prywes

The rising cost of software development has motivated research on the automation of parts of the software development process. Towards that objective, this dissertation presents a non-procedural language called MODEL to describe desired programs of an information processing system. The dissertation also describes a software system that automatically generates conventional business data processing programs from specifications in that language.

Acknowledgements

I would like to express my appreciation to my supervisor Prof. N. S. Prywes for his advice and direction in the research project, and to the Office of Naval Research for contract number N00014-67-A-0216-0014 under which the major part of my research was done.

Thanks are due to Prof. Howard Morgan for his helpful discussions and encouragement.

Appreciation is also expressed to the following graduate students for their programming help in the software implementation: Alex Pelin for his help in the string storage and retrieval sub-system, the cross reference report, and various other subroutines (1974-1975); Stan Cohen for his help in modifying the lexical analyzer of the DDL Project for the MODEL project, and for converting the Berztiss FORTRAN cycle enumeration algorithm into PL/1 (Summer 1974); Barry Soroka for writing several subroutines dealing with input/output code generation (Summer 1975); Jim Coffman for writing several subroutines in the area of matrix analysis, flow optimization, and code generation (Summer 1975); Yung Chang for his aid in precedence determination (Summer 1975).

I would also like to mention Dr. David Sherr with whom I have had several relevant discussions and Dr. Jesus Ramirez with whom I worked on a preceding project (DDL) from which the SAPG and Data Description portion of MODEL were adapted and extended.

I would like to thank Barbara Collins for her typing of the earlier dissertation drafts and various technical reports.

My thanks also go to my family and friends who have encouraged me during this work.

Finally, but most importantly, my deepest gratitude goes to my wife, Judy, not only for her countless hours of keypunching and typing, day and night, but also for her moral support, devotion, and love throughout this project.

INDEX

Adjacency matrix (see also weighted adjacency matrix) 171, 197, 211, 215, 226, 257

ADS 23

Algorithm 11, 12, 14, 88, 316

Ambiguity 12, 92, 199, 200, 202, 210

ANSI_STD 54

Application independent 18, 35

Arithmetic expression 65, 67, 73, 240, 315

Assertion 25, 33, 39, 41, 43, 46, 58, 59, 60, 61, 62, 63, 64, 65, 66, 68, 69, 70, 72, 73, 74, 75, 79, 81, 92, 93, 94, 163, 186, 194, 196, 197, 199, 200, 201, 203, 204, 241, 246, 247, 249, 253, 254, 257, 288, 294, 295, 297, 299, 321, 323, 324

Associative memory 9, 92, 116, 127, 131, 146, 159, 227

Assumption 199, 200, 201

Automate 1, 2, 3, 4, 7, 13, 88

Automatic 95, 314

Automatic programming 26

Automatic sequencing (see also PRECED, topological sorting) 162

Automation 16, 20, 21, 22, 23, 24, 25, 26, 31, 56, 314, 317

Balzer 21

BIN 60

BINARY 60

Block size 56

BLOCKSIZE 54

BNF 101

Boehm 15, 17

Business 1, 2, 3, 27, 31, 279

BYPASS 54

CARD 54, 55

CGSUM 311

CHAR 60, 61

Character 60

Character class 121, 132

Character set 48

CHECEND 230, 259, 260

CHECKDO 230, 259, 260

CHECLAB 230, 262, 263

CHECOND 250

Chief programmer team 16

CHOICE 72, 73, 74, 186, 196, 264, 301, 303

COBOL 19, 31, 52, 229, 270, 279, 316

Code generation 8, 88, 90, 94, 186, 210, 218, 233, 235, 241, 244, 245, 247, 250, 254, 255, 257, 258, 259, 262, 277, 278, 279, 280, 282, 286, 291, 295, 301, 303, 310, 311, 312, 313

Code, coder, coding 1, 2, 3, 4, 10, 20, 24, 26, 31, 32, 88, 94, 118, 218, 277, 279, 301, 302

CODEGEN 277

Compilation, compiler 8, 11, 95, 117, 225, 305, 311, 312, 313, 314

Completeness (see also incompleteness) 8, 88, 92, 121, 314

Computer 1, 16, 21, 22, 29, 212

Computer program 2, 6, 35, 45

Conditional code 250, 252, 295, 297, 298, 299

Conditional function 79, 196, 197, 247, 249, 251, 252, 255, 297

Consistency (see also inconsistency) 8, 88, 92, 121, 314

Control code 264, 295, 298

Control logic 30, 93, 211, 218

Control structure 6, 16, 19, 31, 34, 80, 94, 230, 247, 250, 279

Cost 1, 3, 13, 15, 16, 21

CRADJMT 188, 189

CRDICT 185, 186, 187

Cross reference 90, 92, 112, 119, 159, 160, 161

CRPATHS 205, 206

Cycle 80, 206, 210, 215

Cycle detection 176

Cycle enumeration 206, 207, 208, 209

CYCLES 206, 207, 208

CYL 55

Data base 6, 7, 63

Data base management system 19, 25

Data description, data description language 7, 12, 25, 33, 36, 38, 46, 50, 51, 52, 62, 92, 99, 163, 186, 266, 273, 319, 321

Data independence 19, 36

Data packing 239, 241, 244, 246, 286, 287, 288, 289, 290, 292, 293

Data processing 2, 4, 42, 76, 279, 314, 316

Data structure 240, 241, 245, 263, 266, 270, 272, 273, 279, 285, 286, 288, 305, 308

Debug, debugging 2, 3, 4, 20, 24, 31, 32, 88

Decision rules 72, 321, 323

Declaration 266, 269, 270, 271, 273, 274, 275, 276, 277, 305, 306, 307, 308, 309, 310

DELIM 67, 68, 76, 323

DENSITY 54

DEPSALE 42, 46, 58, 63, 74, 76, 80, 162, 163, 164, 165, 201, 245, 254, 258, 261, 262, 264, 285, 300, 301, 318, 319, 321, 323, 324

Descendant (see also direct descendant, successor) 193, 252

Design 1, 2, 3, 4, 11, 88

DICT 185

Dictionary 166, 170, 176, 185, 186, 187, 230, 235, 246, 253

Digraph, directed graph (see also weighted directed graph) 8, 168, 169, 170, 171, 176, 185, 186, 203, 205, 208, 215, 217, 218

Direct descendant (see also successor, descendant) 166, 203, 213

Directory entry 146, 149, 153, 156

Disjunctive normal form 73, 155

DISK 54, 56

EBNF 47, 83, 92, 99, 101, 102, 111, 112, 119, 131

EBNF/WSC 99, 101, 102, 111, 112, 113, 115, 116, 117, 118, 119, 125, 126, 128, 129, 130, 131, 132, 134

EMPTY 81, 300, 301

Encode, encoding routine 102, 111, 112, 117, 119, 126, 130, 131, 134, 149, 153

ENDPREL 189, 194

ENEXDP 194, 195, 196, 197

ENHRREL 189, 190, 191, 192

ENIMDP 198

ENPTREL 189, 202

Error message, error routine 102, 115, 119, 125, 126, 128, 129, 131, 134, 176, 179, 180, 181, 182, 183, 184, 185, 196, 198, 199, 200, 203, 204, 225

Execution 95, 312, 315

EXIST 58, 59, 64, 65, 66, 186, 196, 208, 240, 241, 261, 288, 291, 299

Field 59, 67, 71, 255

FIELD 59, 60

File 5, 7, 30, 33, 41, 53, 68, 71, 88, 163, 238

File organization 5, 20, 32, 41, 235, 238

FILE 52, 53, 55, 57, 272, 305

Finite state machine 121

FIXED 54, 60

Flag 264, 265, 303

Flowchart 3, 8, 9, 10, 30, 88, 90, 93, 94, 210, 211, 218, 226, 227,
229, 230, 2 66, 267, 268, 279

Flowchart creation 277

Flowchart generation 301

Flowchart report 218

Flowchart table 218, 226, 227, 228, 229, 230, 232, 233, 234, 239, 241,
244, 245, 246, 247, 250, 252, 253, 254, 255, 257,
258, 259, 263, 264, 265, 266, 273, 279, 282, 283,
288, 294, 297, 298, 299, 302, 303, 304

FLOWOPT 219, 220, 221, 222

Floyd 206

FNDISRC 198, 199

FOREACH 34, 71, 72, 219, 223, 225, 253, 261

Function 5, 75, 76, 79, 82, 196, 197, 247, 249, 250, 265, 297, 299, 301, 303, 316, 324

Functional specifications 20, 21, 22, 23, 24, 26, 29, 30, 317

GDCLT 230, 274, 275

GDCLTAB 266

GENDO 299

GENFLT 230, 231, 232, 234, 246, 250, 262

GENGOTO 233

GENIOCD 283

GENLAB 253

GENPL1 279, 281

GFLTRPT 266

Gibb 23

GIMFLD 302

GMODCD 282

GPL1DCL 305, 306, 311

GPROCCD 294, 295, 298, 301

Grammar (see also EBNF) 14, 83, 102, 114, 128, 129

Graph (see also directed graph, network creation) 10, 206, 212

Graph analysis 176, 177, 185, 202, 226

Graph theory 88, 93, 162, 211

GROUP 58, 59

Header 38, 46, 49, 63, 319

Hierarchical relationship 166, 172, 189, 190, 191, 192, 193, 194

Housekeeping routine 133, 134, 264

IBM 17, 60, 76

IBM_STD 54

IDASSN 230, 246, 247, 248, 253, 304

IDEND 233

IDFIN 233

IDFLDAS 230, 255, 256

IDGOTO 265

IDIOCD 230, 234, 235, 236, 264, 304

IDMODNM 233

IDPACK 239, 241, 242, 243

IDRSET 233, 265

Implementation 12, 27, 31

Implicit predecessor 198, 200, 257, 302

Incompleteness (see also completeness) 12, 176, 188, 193, 196, 198, 202, 203, 210

Inconsistency (see also consistency) 12, 176, 188, 193, 198, 202, 210, 225

Indexed sequential (ISAM) 5, 42, 51, 55, 56, 203, 235, 238, 316

INDEXED, INDEXED_SEQUENTIAL 55

Information processing system, Information system 1, 3, 24, 27, 29, 31, 314

Input/output 234, 235, 237, 240, 272, 273, 279, 283, 284, 286, 305, 308

INT_NAME 54

Integer 48

Inter-file relationship (see also POINTER) 41, 62, 63

Interact, interactive 90, 316

Interim 41, 61, 199, 200, 203

INTERIM 61

Invariant 223

ISDOS 20, 23, 25, 26

Iteration 5, 93, 94, 211, 219, 224, 225, 253, 254, 259, 260, 261, 299

Key 53, 64, 155

KEY 52

Label, label table 262, 263

Leavenworth 33

LEN 60, 66, 67, 186, 196, 208, 240, 241, 288

LEX 118

Lexical analysis, analyzer 119, 121, 132, 159

Library 36, 45, 75, 76, 316, 324

Line_length 54

Logical expression 68, 73, 155, 156, 315

Loop 219, 223, 224, 253, 259, 260, 262, 285, 299, 303

Lr-1 114, 115, 128

Machine independence 6, 35

Machine language 313

Management 1, 2, 3, 17, 19

MAX_BLOCKSIZE 54

MAX_RECORDSIZE 54

MERGPL1 249, 311

MINSALE 163, 234, 239

MIT 26

MODEL 26, 31, 32, 33, 45, 53, 56, 60, 61, 63, 314, 315, 316, 323

MODEL language 2, 4, 5, 6, 7, 8, 14, 24, 25, 27, 34, 36, 42, 80, 82, 83, 88, 90, 92, 99, 101, 111, 113, 115, 116, 121, 125, 128, 132, 133, 134, 146, 149, 159, 162, 170, 201, 240, 266, 273, 285, 286, 291, 300, 317, 318, 319, 321, 324

MODEL processor 4, 7, 9, 11, 12, 25, 27, 32, 36, 38, 75, 79, 83, 88, 90, 93, 95, 98, 113, 121, 159, 162, 176, 194, 198, 199, 200, 204, 210, 215, 218, 266, 277, 279, 291, 312, 313, 314, 315, 316, 317, 318, 324

MODEL specification 126, 127, 162, 163, 166, 168, 170, 172, 176, 185, 186, 188, 194, 201, 204, 224, 249

Modular, modularity 16, 34, 94

Module 2, 3, 4, 5, 7, 8, 10, 11, 14, 17, 20, 23, 24, 25, 30, 31, 36, 38, 41, 45, 49, 50, 68, 76, 88, 90, 93, 95, 226, 233, 282, 313, 314, 319, 321

MODULE 49

Natural language 21, 316, 317

Network analysis (see graph analysis) 185

Network creation 176, 177, 179, 185

NEW 70, 71, 72, 186, 257, 302, 308

No_trks 54

Node 170, 171, 172, 185, 208, 211, 212, 213, 215, 218, 219, 223, 226, 227, 229, 230, 2 34, 239, 250, 251, 252, 253, 255, 261, 263, 264, 297, 298, 301

Non-procedural 2, 3, 4, 6, 7, 8, 11, 24, 25, 27, 33, 34, 56, 60, 75, 81, 88, 113, 146, 314, 315

Non-terminal 101, 117, 133

NONE 54

NUMERIC 60, 61

Nunamaker 23, 25

OLD 70, 71, 72, 81, 186, 257, 302, 308

Operating system 25, 90, 305

Optimization 312, 313

Optionality 291

Order independence 34, 60

Order vector 213, 215, 217, 218, 219, 225, 226, 227, 229, 230, 262, 264

ORG 22, 54

Packages 17, 18, 315

PACKINF 244, 246

Parallel processor 218

PARITY 54

Path matrix 205, 206, 226

Physical design 20, 24, 26, 30, 317

P1/1 6, 9, 10, 11, 31, 75, 76, 90, 94, 95, 113, 117, 118, 226, 227, 229, 233, 234, 240, 241, 245, 246, 247, 252, 254, 258, 259, 262, 266, 270, 272, 273, 277, 279, 281, 282, 283, 284, 285, 286, 288, 291, 294, 295, 298, 299, 302, 305, 307, 308, 309, 310, 311, 312, 315, 316

PL1DCL 279, 311

PL1EX 279, 311

PL1OBJ 311

PL1ON 279, 311

PL1PROC 279, 311

POINTER 63, 64, 81, 186, 201, 208, 223, 285

Pointing relationship (see also inter-file relationship) 166, 189, 201, 203

Pre-processors 18

PRECED see also topological sorting, automatic sequencing) 212, 213, 214

Precedence, precedence graphs, precedence relationship 9, 10, 92, 93, 166, 167, 168, 170, 172, 174, 175, 176, 188, 190, 193, 196, 202, 211, 213, 226, 277

Precedent, predecessor 166, 171, 172, 194, 196, 198, 199, 200, 212, 251, 255

PRINTER 54

Problem domain 21

Problem statement language (PSL) 22, 23, 26, 32, 56

Procedure, procedural 18, 19, 25, 254, 294, 295, 298, 314

Processor (see MODEL) 314

Production rule 101, 112, 114, 115, 117, 129, 130, 131, 133

Program design 20, 24, 26, 27, 30, 32, 93, 211, 277, 313

Program generation 26, 88, 95, 229

Program, programmer 1, 2, 3, 10, 12, 16, 18, 19, 30, 88, 90, 94, 225, 277, 315

Programming language 6, 15, 16, 19, 94, 226

Prywes 15, 20

PUNCH 55

Qualify, qualifier 70, 71, 186, 273

Rank 213, 215, 217

Rank set 213, 215, 216, 218

Rank vector 213, 215, 216, 225

Reachability matrix (see path matrix) 205

Record 53

Record description 57

Record format 56

Record size 56

RECORD 57, 59, 273, 308

Recursive 190

Reference manual 46

Relationship 166

REPLACE 76, 77, 80, 81, 82, 210, 247, 253, 254, 300, 301, 302, 303

REPORT 53

REPORT_ENTRY 53

Reserved words 47, 72

RETRIEVE 146, 149, 155, 156, 157, 159, 160, 235

Sammet 33

SAP 102, 112, 116, 117, 118, 119, 125, 126, 127, 130, 131, 149

SAPG 9, 90, 92, 99, 101, 102, 111, 112, 113, 114, 115, 116, 117, 118, 119, 126, 127, 128, 146, 159

Scope 93, 211, 219, 223, 224, 225, 259, 299

SELECTED 68

Selection criteria (see also subset selection) 33, 41

Semantics, semantic analysis 83, 90, 99, 102, 112, 119, 121, 125, 130, 134

Sequence 176, 211

SEQUENCE 52

Sequencing algorithm 211, 214

Sequential 5, 23, 42, 51, 56, 235, 238, 285, 291, 316

SEQUENTIAL 55

SERIAL# 76, 77

Software 1, 11

Software development (see also systems building) 1, 2, 3, 11, 13, 15, 16, 17, 20, 22, 26, 27, 88, 314, 317

Software system 88

Source 33, 38, 41, 46

Source file 253, 303, 304, 319

SOURCE FILES 49

SOURCE 62, 79

SPACE 54

Specification 2, 3, 8, 13, 30, 88, 90, 314

Stacking 254

START_FILE 54

Statements 38, 88

Storage 54

Storage and retrieval 131, 146, 159

Storage entry 146, 147, 148, 149, 153, 155, 156

Storage medium 41, 53, 55

STORAGE 52, 55

STORE 146, 149, 153, 154, 159

Storing routine 112, 116, 119, 127, 134

Structured programming (see also top-down programming) 16, 17, 31

Subset selection 68, 303, 304

SUBSET 68, 186, 253, 303, 304

Successor (see also direct descendant) 166, 170, 171, 172, 257

SUM 76, 78

Syntax 8, 47, 83, 92, 127, 229, 311, 315

Syntax analysis 90, 92, 99, 102, 112, 113, 121, 126, 277

System analyst 2, 4, 27, 31

Systems building (see also software development) 15, 16, 20

TAG 23

TAPE 54, 55

TAPE_LABEL 54

Target 33, 38, 41, 46

Target file 319

TARGET 50, 62, 79

Teichroew 2, 15

Terminal 117, 129, 132

TERMINAL 54, 56

Testing 20, 31, 32

TODAY 76, 78

Token 114, 115, 117, 119, 125, 131

Top-down programming (see also structured programming) 16, 31

Topological sorting (see also PRECED, automatic sequencing) 212, 214

TRACKS 55

UNDEFINED 54

UNIT 54

University of pennsylvania 26, 99

User requirements 18, 20, 21, 22, 29

Value dependency relationship 166, 189, 194, 195

Variable length 52, 66, 239, 240, 241, 245, 285, 286, 288, 291, 321,
323

Variable repetition 64, 239, 240, 288, 291, 299, 321, 285

VARIABLE 54

VARIABLE_SPANNED 54

Very high level language (see also non-procedural) 27, 315

VOL_NAME 54

Warshall's algorithm 205

Weighted adjacency matrix 168, 171, 172, 173, 176, 185, 188, 192, 194, 196, 198, 202, 204, 210, 211, 213, 250, 255, 257

Weighted directed graph 168, 169, 170

WRDCL 311

WRPL1 310

XPL 113

XREF 160

TABLE OF CONTENTS

1. Introduction 1

1.1 Summary of Research 1

1.1.1 Goal and Purpose of Research 1

1.1.2 Overview of the Non-Procedural Language: MODEL 5

1.1.3 Overview of the MODEL Processor 7

1.2 Summary of Contributions 11

1.3 Organization of This Dissertation 13

2. Background, Movivation, and Survey of Related Literature and Research 15

2.1 Introduction 15

2.2 Improvements to the Systems Development Process 16

2.3 Research on Automation of the Systems Building Process 20

3. The MODEL Language 27

3.1 Introduction 27

3.1.1 Overview of Purpose and Usage 27

3.1.2 MODEL Language Components and Novel Features 33

3.1.3 Language Sections 38

3.1.4 An Example of an Application of MODEL:

The DEPSALE Problem 42

3.2 User Reference Manual for MODEL 46

3.2.1 General Information 46

3.2.1.1 Syntax Notation 47

3.2.1.2 Names 47

3.2.1.3 Character Set 48

3.2.1.4 Integers 48

3.2.2 Specification Header 49

3.2.2.1 Module Name Statement 49

3.2.2.2 Source Files Description 49

3.2.2.3 Target Files Description 50

3.2.3 Data Description 50

3.2.3.1 File Statement 52

3.2.3.2 Storage statement 54

3.2.3.3 Record Description 57

3.2.3.4 Group Description 58

3.2.3.5 Field Description 59

3.2.3.6 Interim Data Description 61

3.2.4 Assertions 62

3.2.4.1 Inter-file Relationships: POINTER Assertions 63

3.2.4.2 Assertions for Variable Repetition:
EXIST Assertions 64

3.2.4.3 Assertions for Variable Length: LEN Assertions 66

3.2.4.4 Subset Selection: SUBSET Assertion 68

3.2.4.5 Assertions for Computation Rules 69

3.2.4.6 Assertions for Decision Rules:
CHOICE Assertions 72

3.2.4.7 Use of Functions 75

3.2.4.8 Allowable Cycles: The REPLACE Function 80

3.3 The MODEL Language Described in EBNF 83

4. The MODEL Processor 88

4.1 Overview 88

4.2 Syntax and Statement Analysis 99

4.2.1 EBNF, SAPG and the Syntax
Analysis Program 99

4.2.1.1 Specification of MODEL
Using EBNF and the SAPG 99

4.2.1.2 Adequacy and Limitations of SAPG 113

4.2.1.3 How the SAPG Produces the SAP 116

4.2.2 Supporting Subroutines for EBNF of MODEL 119

4.2.2.1 The Lexical Analyzer 119

4.2.2.2 Statement Semantic Analysis 121

4.2.2.3 Error Message Stacking Routines 125

4.2.2.4 Encoding User Statements 126

4.2.2.5 Statement Storage Routines 127

4.2.3 Experience with and Use of EBNF/WSC and SAPG 128

4.2.4 The String Storage and Retrieval Sub-System 146

4.2.4.1 Introduction 146

4.2.4.2 The Directory Storage System 146

4.2.4.3 Storage Entries Format and Tables for MODEL Statements 149

4.2.4.4 The STORE Procedure 153

4.2.4.5 The RETRIEVE Procedure 155

4.2.5 Components of a General System for Statement Analysis, Storage and Retrieval 158

4.2.6 Cross-Reference and Attribute Report 159

4.3 Analysis of MODEL Specification 162

4.3.1 Introduction and Background 162

4.3.2 Overview of Sub-phases in Network Creation and Analysis 176

4.3.3 Sub-phases of Network Creation and Analysis 185

4.3.3.1 Creating a Dictionary for Row and Column Numbers of the Weighted Adjacency Matrix 185

4.3.3.2 Creating the Weighted Adjacency Matrix and Entering Precedence Relationships within it 188

4.3.3.3 Entering Hierarchical Relationships 190

4.3.3.4 Entering Value Dependency Relationships 194

4.3.3.5 Entering Conditional Dependency Relationships 197

4.3.3.6 Finding Implicit Predecessors 198

4.3.3.7 Entering Pointing Relationships 201

4.3.3.8 Graph Analysis of Adjacency Matrix 202

4.3.3.9 Path Matrix Creation; Cycle Detection and Enumeration 205

4.4 Automatic Program Design and Determination of Sequence and Control Logic 211

4.4.1 Precedence Determination and Sequencing Algorithm 211

4.4.2 Scope and Iteration Analysis and Optimization 219

4.4.3 Generation of Flowchart Table 226

4.4.3.1 Control Module for Generating Flowchart Table 229

4.4.3.2 Identifying Module Name 233

4.4.3.3 Identifying Input/Output Commands 234

4.4.3.4 Identifying Data Packing and Unpacking Information for

Variable Repetition and Length 239

4.4.3.5 Identifying Assertions Information 246

4.4.3.6 Identifying Field Associations 255

4.4.3.7 Checking for Iterations 259

4.4.3.8 Checking for Label Generation 262

4.4.3.9 Identifying Housekeeping and End-of-Program Tasks 264

4.4.3.10 The Flowchart Table Report 266

4.4.3.11 Generating Table of Data Structures Declarations 266

4.5 Code Generation 277

4.5.1 Generation of PL/1 Program 277

4.5.1.1 Generating the Procedure Heading Code 282

4.5.1.2 Generating Input/Output Code 283

4.5.1.3 Generating Code for Variable Length, Optional, and Variably Repeating Data 285

4.5.1.4 Generating Procedures Code 294

4.5.1.5 Generating Conditional Code 296

4.5.1.6 Generating Iterative Code 299

4.5.1.7 Generating Replacement-Stacking Code 300

4.5.1.8 Generating Code for Implicit Field Assignments 302

4.5.1.9 Generating Code for Resetting Switches and Flags 303

4.5.1.10 Generating Code for Subsets 303

4.5.1.11 Generating PL/1 Declarations 305

4.5.1.12 Other Code Generation Supporting Routines 310

4.5.2 Code Generation Summary 311

4.6 Compilation and Execution of the Generated Program 312

5. Conclusions 314

Appendix A: An Example of the Use of MODEL 318

Appendix B: PL/1 Source Code 347

LIST OF FIGURES

Figure 3.1. Stages in Software Development Project 28

Figure 3.2. Program and Data Independence in MODEL Automatic Program Generation System 37

Figure 3.3. Components of a MODEL Specification 45

Figure 3.4. Illustration of Department Store Sale (DEPSALE) 44

Figure 4.1. Overview of MODEL Processor 89

Figure 4.2. Phases of the MODEL Processor 91

Figure 4.3. Major Modules of the MODEL Processor 96

Figure 4.3a. Index of Major Modules 97

Figure 4.4. Block Diagram of SAPG and SAP 100

Figure 4.5. EBNF/WSC for MODEL 103

Figure 4.6. More Detailed View of SAPG and SAP With Supporting Subroutines 120

Figure 4.7. Sample Directory and Storage Entries 150

Figure 4.7a. Example of Retrieval Mechanism for Keys B and G 158

Figure 4.8. Example of Cross Reference and Attribute Report 161

Figure 4.9. Sample Set of MODEL Statements 164

Figure 4.10. Digraph for MODEL Specification of Figure 4.9 169

Figure 4.11. Weighted Adjacency Matrix for Sample Set of MODEL Specification 173

Figure 4.12. Cycle Enumeration of a Sample Digraph 209

LIST OF FIGURES (continued)

Figure 4.13. Digraph Nodes Sequenced in Precedence Order 217

Figure 4.14. Generating Object Program after Analysis and Precedence Phases 228

Figure 4.15. Components of Generating the Flowchart Table 232

Figure 4.16a. Flowchart Table Report Entry Types 267

Figure 4.16b. Sample Flowchart Report 268

Figure 4.17. Overview of Generation of Data Structures Declarations 269

Figure 4.18. Overview of the Code Generation Phase 278

Figure 4.19. Components of Generating PL/1 Code 280

Figure A1. Illustration of Department Store Sale (DEPSALE) 320

Figure A2. Data Network for DEPSALE Problem 322

Figure A3. Listing of DEPSALE Problem in MODEL 325

Figure A4. Directed Graph of DEPSALE Problem 346

LIST OF TABLES

Table 4.1. Character Classes for MODEL Language 122

Table 4.2. State Transition Matrix for MODEL Lexical Analyzer 123

Table 4.3. Lexical Analysis Actions 124

Table 4.4. Storage Entries Format for MODEL 151

Table 4.5. Precedence Types 167

Table 4.6. Illustration of Precedence Relationships of Table 4.5 for Matrix of Figure 4.11 174

Table 4.7. Steps in Network Creation and Analysis 177

Table 4.7a. Error Conditions Detected during Network Creation and Analysis 178

Table 4.7b. Summary of Errors Detected during Entire Matrix Analysis Phase 180

Table 4.8. Data Declarations Table 271

Table 4.9. Input/Output Transformations from Flowchart Table to PL/1 284

Table 4.10. Generation of PL/1 File Declaration 307

Table 4.11. Generation of PL/1 Hierarchic Record Structure 309

LIST OF ALGORITHMS

Algorithm Name	Algorithm Page no.	PL/1 Code Page no.
STORE	154	428
RETRIEVE	157	424
CRDICT	187	352
CRADJMT	189	351
ENHRREL	191	363
ENEXDP	195	359
ENPTREL	202	367
CRPATHS	206	357
CYCLES	207	358
PRECED	213	422
FLOWOPT	221	368
GENFLT	231	384
IDIOCD	236	404
IDASSN	248	401
IDFLDAS	256	400
CHECKDO	260	349
CHECEND	260	348
CHECLAB	263	348
GDCLT	274	373
GENPL1	281	350
GMODCD	282	391
GENIOCD	284	385
GENPACK	287	389

LIST OF ALGORITHMS (continued)

Algorithm Name	Algorithm Page no.	PL/1 Code Page no.
PACK	290	414
GPROCCD	295	398
GENDO	299	382
GENEND	299	383
GIMFLD	302	391
GENRSET	303	390
GPL1DCL	306	392
WRPL1	310	440
WRDCL	311	439
MERGPL1	311	421

System Functions:

Algorithm Name	Algorithm Page no.	PL/1 Code Page no.
DELIM		358
REPLACE	300	423
SERIAL#		427
SUM		431
TODAY		431

Bibliography

[ADA 72] Adams, D.L. and Mularkey, J.F., "A Survey of Audit Software", Journal of Accountancy, Sept. 1972.

[ADS 68] ADS, National Cash Register Co., 1968.

[BAK 72a] Baker, F. T., "Chief Programming Team Management of Production Programming", IBM System Journal, No. 1, 1972.

[BAK 72b] Baker, F. T, "System Quality Through Structured Programming", AFIPS Proceedings Fall Joint Computer Conference, 1972, pp. 339-343.

[BAK 73] Baker, F. T. and Mills, H. D., "Chief-Programmer Teams", Datamation, Dec. 1973.

[BAL 72] Balzer, R. M., "A Global View of Automatic Programming", also Memorandum on "Automatic Programming", USC Information Sciences Institute, Marina del Rey, California, Sept. 1972.

[BER 71] Berztiss, A. T., Data Structures Theory and Practice, Academic Press, New York, 1971.

[BOE 73] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973.

[BLO 73] Blosser, P., Konsynski, B. Jr., and Nunamaker, J. F. Jr., "A Model of the Software Development Process for Automatic Code Generation", ISDOS Working Paper #102, May 1973.

[CAR 73] Cardenas, A.F., "Evaluation and Selection of File Organization -- a Model and System", Communications of the ACM, Sept. 1973, pp. 540-548.

[COD 71a] CODASYL Systems Committee Technical Report, Feature Analysis of Generalized Data Base Management Systems, May 1971.

[COD 71b] CODASYL Data Base Task Group Report, ACM, 1971.

[COD 70] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, 13, 6, June 1970.

[CON 63] Conway, M. E., "Design of Separable Transition Diagram Compilers", Communications of the ACM, July 1963.

[COU 73] Couger, J. D., "Evolution of Business Systems Analysis Techniques", Computer Surveys, Vol. 5, No. 3, Sept. 1973.

[DAH 72] Dahl, Dijsktra, and Hoare, Structured Programming, Academic Press, 1972.

[DON 73] Donaldson, J. R., "Structured Programming", Datamation, Dec. 1973.

[FLO 67] Floyd, Robert W., "Non-Deterministic Algorithms", Journal of the ACM, Vol. 14, No. 4, pp. 636-644, Oct. 1967.

[FRE 72] French, A., "A Syntax Analysis Program Generator", Masters Thesis, Computer and Information Sciences Department, Moore School of Electrical Engineering, University of Pennsylvania, 1972.

[GIB 75] Gibb, K. R., "Automatic File and Module Design", Internal Memo, Bell Laboratories, Jan. 1975.

[GRA 73] Graham, Clang, and DeVeney, "A Software Design and Evaluation System", Communications of the ACM, Feb. 1973.

[GRA 72] Grape System User's Manual, Vol. 2, E. I. Du Pont de Nemours and Co. Inc., July 1972.

[HAX 75] Hax, A. C. and Martin, W. A., "Automatic Generation of Customized Model Based Information Systems for Operations Management", *Proceedings on the Wharton Conference on Research on Computers in Organizations,* Philadelphia, Oct. 75, H. L. Morgan, ed., pp. 117-121.

[HER 73] Hershey, Rataj, Teichroew, and Berg, PSL Manual, ISDOS Working Paper #68, University of Michigan, Oct. 1973.

[HEW 71] Hewitt, C., "Planner: A Language for Proving Theorems in Robots", *International Joint Conference on Artificial Intelligence,* London, 1971.

[KAH 62] Kahn, A., B., "Topological Sorting of Large Networks", *Communications of the ACM,* Vol. 5, pp. 558-562, 1962.

[KOS 74] Kosy, D. W., "Air Force Command and Control Information Processing in the 1980's: Trends in Software Technology", United States Air Force Project Rand, June 1974.

[LAN 65] Langefors, B., "Information System Design Computations Using Generalized Matrix Algebra", BIT 5 2, 1965.

[LEA 74] Leavenworth, B. M. And Sammet, J. E., "Overview of Non-Procedural Languages", *SIGPLAN - Proceedings of a Symposium on Very High Level Languages,* Mar. 1974.

[MAR^74] Martin, W. A., "Automatic Programming Group Progress Report July 1973^-^July 1974", Massachusetts Institute of Technology, 1974.

[McK 70] McKeeman, W. A., Horning, J. J., and Wortman, D. B., *A Compiler Generator,* Prentice Hall, 1970.

[MEF 74] Merten, A. G. and Fry, J. P., "A Data Description Language Approach to File Translation", Data Translation Project, University of Michigan, Apr. 74.

[MIL 71] Mills, H., "Top Down Programming in Large Systems," Debugging Techniques in Large Systems, Randall Rustin, Ed., Prentice Hall, 1971, pp. 41-45.

[MIL 73] Miller, E. F. and Lindamood, G. E., "Structured Programming", Datamation, Dec. 1973.

[NUN 69] Nunamaker, J. F. Jr., "On the Design and Optimization of Information Processing Systems", Ph.D Dissertation, Case Western Reserve University, June 1969.

[NUN 71] Nunamaker, J. F. Jr., "A Methodology for the Design and Optimization of Information Processing Systems", AFIPS Proceedings, 1971, SJCC, pp. 283-294.

[NUN 72a] Nunamaker, J. F. Jr., et al, "Processing Systems Optimization Through Automatic Design and Reorganization of Program Modules", CSD TR77, Purdue University, Dec. 1972.

[NUN 72b] Nunamaker, J. F. Jr., et al, SODA, ISDOS Working Paper #36, University of Michigan, Feb. 1971.

[PAR 72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Dec. 1972.

[PL1 75] PL/1 Reference Manuals, IBM Publication, 1975.

[PRO 74] "Proceedings of a Symposium on Very High Level Languages", SIGPLAN Notices, Mar. 1974.

[PRY 66] Prywes, N. S., "Man-Computer Problem-Solving with Multi-List", Proceedings of IEEE, Vol. 54, No. 12, Dec. 1966, pp. 1788-1801.

[PRY 74] Prywes, N. S., "Automatic Generation of Software Systems -- a Survey", Data Base, Vol. 6, No. 2, Fall 1974.

[PRY 75] Prywes, N. S., "Final Report, Automatic Computer Program Generation for Automatic Testing Systems (ATS)", U.S. Army Armament Command, Frankford Arsenal, Philadelphia, June 1975.

[PRY 72] Prywes, N. S and Smith, D. C. P., "Organization of Information", Annual Review of Information Science Technology, Vol. 7, ed. Carlos H. Cuadro, 1972.

[RAM 73] Ramirez, J. A., "Automatic Generation of Data Conversion Programs Using a Data Definition Language", Ph.D. Dissertation, Moore School of Electrical Engineering, University of Pennsylvania, 1973.

[RAM 74] Ramirez, Rin, and Prywes, "Automatic Generation of File Conversion Programs Using a Data Description Language", Proceedings of ACM-SIGFIDET, May 1974.

[RIN 74] Rin, N. A. and Brown, M., "An Overview of a System for the Automatic Generation of File Conversion Programs", Software -- Practice and Experience, Mar. 1974.

[RIN 75] Rin, N. A., "Documentation Manual for the MODEL Processor", Internal Technical Report, Computer and Information Sciences Dept., Moore School of Electrical Engineering, University of Pennsylvania, Aug. 1975.

[RUT 74] Ruth, G. R., "Status of Protosystem I", Automatic Programming Group, Massachusetts Institute of Technology, Mar. 1974.

[SEV 72] Severance, D. G., "Some Generalized Modeling Structures for Use in Design of File Organizations", Ph.D. Dissertation, University of Michigan, 1972.

[SHA 73] Shapiro, B. A., "A Survey of Problem Solving Languages and Systems", TR-235, Office of Computing Activities, University of Maryland, Mar. 1973.

[SHE 74] Sherr, D. M., "A Proposed Non-Procedural Language for Structured Systems Development", SIGBDP, 1974.

[SIB 73] Sibley, E. H. and Taylor, R. W., "Data Definition and Mapping Language", Communications of the ACM, Vol. 16, No. 12, Dec. 1973.

[SMI 71] Smith, D. P., "An Approach to Data Description and Conversion", Ph.D. Dissertation, Computer and Information Sciences Department, Moore School of Electrical Engineering, University of Pennsylvania, 1971.

[TAG 68] TAG Sales and Systems Guide, IBM, GY20-0358-1, 1968.

[TEI 71] Teichroew, D. and Sayani, H., "Automation of Systems Building", Datamation, Aug. 1971.

[TEI 72] Teichroew, D., A Survey of Languages for Stating Requirements for Computer- Based Information Systems", Fall Joint Computer Conference, 1972.

[TEM 72] Teichroew, D. And Merten, A., "The Impact of Problem Statement Languages on Evaluating and Improving Software Development Performance", Fall Joint Computer Conference, 1972.

[THA 71] Thall, R. M, "A Manual for PSA/ADS: A Machine-Approach to Analysis of ADS", ISDOS Working Paper #35, University of Michigan, Oct. 1971.

[WAR 68] Warshall, S., "A Theorem on Boolean Matrices", Journal of the ACM, Vol. 9, pp. 11-12, 1962.

[WIL 72] Will, H. J., "Computer-Based Auditing", Canadian Chartered Accountant, Feb. 1972.

[WIN 72] Winograd, R., Understanding Natural Language, Academic Press, Massachusetts Institute of Technology, 1972.

[YOU 58] Young, J. W and Kent, H. K, "Abstract Formulation of Data Processing Problems", National Cash Register Co., also given at 13th ACM National Meeting, June 1958.

[YOU 65] Young, J. W., "Non-Procedural Languages -- a Tutorial", National Cash Register Co., Mar. 1965.

CHAPTER 1

INTRODUCTION

1.1 Summary of Research

1.1.1 Goal and Purpose of Research

The ultimate objective of the research reported in this dissertation is the automation of the software development process. The literature on software development of the past several years (see literature survey in Chapter 2) has recognized the unlikelihood of continued building of large complex software systems by the conventional manual methods because of rising software development costs and demand on the one hand and the shortage of sufficient trained personnel on the other. This research and that of others (surveyed in Chapter 2) are demonstrating the feasibility of reducing the costs of software development by utilizing the computer itself to produce industrial software systems automatically based on functional specifications from the business or management specialist. The long term objective of the research area reported here is to eliminate the computer-trained personnel involved in the physical system design of the information processing system, the computer program designer, and the coder as middle men between the non-computer trained business specialist who can specify the functional automation requirements formally and the desired operational application software system. The direction of such research is to make computer programming not only much easier and less costly and reduce the dependence on computer programmers in software development, but ultimately to eliminate the physical system designer, the application programmer, and coder by automatically producing computer programs for the business or management specialist on demand.

Toward this objective, the immediate goal of this dissertation has been the development of

(1) a non-procedural Module Specification Language (MODEL) in which an analyst specifies a functional module of an application system; and

(2) a software system that automates a significant portion of the software development process; namely, the program module design, coding, and debugging phases of a large class of conventional data processing programs to be described.

In order to pinpoint the area of automation covered by this research, the following discussion first delineates the stages of the conventional software development process of a typical software development project. Although the delineation of these stages has been viewed differently in the literature (surveyed in Chapter 2), they are generally as follows:

(1) Determination and Statement of Overall Problem Requirements.

(2) Analysis of Problem Requirements and Production of System Functional Specifications which describe the information that are input to and output from the envisioned system as a whole.

(3) System Physical Design, which includes identification and specification of component program modules, their individual inputs and outputs, and evaluation and selection from alternative file

structures (the term "program module" here is used loosely here to refer to a functional sub-component of the system).

(4) Program module detailed design, conventionally by such means as program flowcharts.

(5) Program module programming, coding, and debugging, usually in a high-level programming language which in turn is compiled into a machine language program.

(6) Integration, Operation, and Maintenance

Documentation of the system is a process parallel to all stages above. Each one of the above stages in the software development process depends upon the previous one above it and the stages utilize specialists in management, business, information systems analysis, design and programming.

As indicated, future demand and costs of analysis, design, and programming, will necessitate the automation of several of these phases. Other related research projects and literature have provided system development tools and have made inroads into the partial automation of some of these software development processes. These include the various automated aids to the system analyst. Software development automation to date, however, has been primarily in the analysis and design phases, and to date there has been no general-purpose and application-independent automatic system that produces complete programs from non-procedural functional specifications.

An initial goal of this research was the development of the Module Description Language (MODEL), a very high level and purely non-procedural language in which a business systems analyst could describe the functional modules of a desired application information system.

More importantly, the goal of this research is the automation of stages 4 and 5 above, the program design, coding, and debugging phases of software development, by designing a software system to process MODEL specifications and produce desired programs. The MODEL user could utilize the MODEL language and Processor in developing an application system, by writing a formal MODEL specification for each of the modules in the application system after the system analysis and design phases above it are completed by some other process. These descriptions would then be submitted to the MODEL Processor. The MODEL Processor performs the program writing task by automatically completing the program design phase and generating the program, thus replacing the coding, debugging, and testing phases for a wide class of data processing problems described below.

The class of programs which the MODEL Processor would be capable of generating automatically from non-procedural specifications is one whose programs have traditionally been widely developed manually. The majority of programs traditionally written in the data processing environment can be generated by the MODEL Processor. The following are some of the features by which this class of programs is characterized. Within each of the programs generated by MODEL, there is a major loop in which processing is done and records are read or written one at a time. Within each iteration through the major program loop, however, records can be read from any number of files, routine processing functions can be invoked, inner iterations through repeating items take place, data is moved to output areas, records are written, etc. The programs generated by MODEL process one record at a time but can preserve information (such as totals) across records by use of functions. This class encompasses so-called "Transaction Processing" programs that are widely written. The programs generated by MODEL can process data from any number of input files and produce any number of output files, whose file organizations are either sequential or indexed sequential. There are programs currently outside the realm of this class, but the techniques described in this dissertation could and are being expanded to other areas. Other capabilities and restrictions of MODEL can be found in Chapter 3.

1.1.2 Overview of the Non-procedural Language: MODEL

The MODEL language allows the specifier to describe the desired program module by a set of statements describing existing (or "source") data, desired (or "target") data, and a set of statements called "assertions" which describe various types of data interrelationships. The use and components of the MODEL language specification are explained in detail in Chapter 3.

Some of the novel features that were incorporated into the language design are summarized here. The most apparent new feature of MODEL when compared to programming languages or to command languages of data base management systems is its non-procedural nature. The MODEL language is designed to be as close to non-procedural as possible in several respects. All statements are declarative or descriptive as opposed to imperative. The statements describe only data or data relationships to other data. The procedural detail of programming languages (with statements such as READ, WRITE, MOVE, OPEN, CLOSE, etc.) is not in the MODEL language, since actions are deduced by the Processor. Furthermore, the statements consist of independent descriptions and can be submitted in any order. Therefore the specification can be built modularly with the statements added in independent stages and in any order. There are no control structures connecting them, since the sequencing and control logic code is produced automatically by the Processor. Finally, there is a total absence of computer programming terminology and concepts. There is no reference to sequences of operations, control code, input and output commands, counting, data movement, memory, computer implementation, or any other processes.

Another feature of the MODEL language is its intended use for automatic generation of the application programs by the Processor into a high-level compiler language, namely PL/1, rather than into a particular machine language. This promotes machine independence and transferability.

MODEL furthermore is domain/application independent since it deals solely with information and information relationships. Since the MODEL Processor has only computer programming knowledge (i.e. a "model" of programming), it should be applicable to a broad spectrum of applications.

A final concept of MODEL is a potential capability to store, control, and share centrally-maintained data descriptions and a method to provide data and program independence without resorting to the more common but less efficient method of execution-time binding of a data description to the generated program. The sharing of a centrally-maintained data description would be especially useful in a shared data base environment. This feature is described conceptually further in Chapter 3, but is not included as part of the Processor described in Chapter 4. It also suggests a capability of automatic monitoring of changes to the data base description and automatic invocation of the MODEL Processor itself by a monitor to regenerate affected modules.

1.1.3 Overview of the MODEL Processor

Chapter 4 describes a software system called the MODEL Processor, which would perform the program writing function. The MODEL Processor has been designed in order to "model" and automate the program module design, coding, and debugging phases of software development based on module specifications in the non-procedural MODEL language. It presupposes that the functions of the desired application have been described to the extent that the file, inter-file, and intra-record structures have been described and that the system has been partitioned into functional modules. A module is then formally described and specified in the MODEL language, whose statements are then submitted to the MODEL Processor. The MODEL Processor, in turn, performs the following tasks:

(1) analysis of the specification for completeness and consistency;

(2) module design, which includes generating a flowchart-like sequence of events for the module; and

(3) code generation function, thus replacing the tasks of the application programmer/coder.

Although some aspects of the MODEL Processor have some similarity to the conventional compiler (e.g. in syntax analysis and code-generation), it differs greatly in its capability to process a non-procedural specification language described above, in its internal model of data processing programming, in its ability to detect logical inconsistencies and incompleteness, and in its ability to make assumptions about the program logic. These capabilities are partially based on an application of graph theory. Relationships within the specification are detected by the MODEL Processor and represented in a directed graph, which is used as a basis for analysis, design and the program generation task.

Another important function of the MODEL Processor is to interact with the specifier to indicate necessary supplements or changes to the submitted statements. Messages are sent to the user to indicate missing, inconsistent, or ambiguous statements and, when possible, suggestions for remedying them.

The Processor would produce a complete PL/1 program ready for compilation, and various reports concerning the specification and the generated program, such as a listing of the specification, a cross-reference report, a flowchart-like report on the generated program, and a listing and summary of the generated program. These are regenerated whenever a specification is submitted.

The MODEL Processor goes through five phases in its analysis, design, and program generation tasks. In the first phase, the provided MODEL Specification is analyzed to detect syntactic errors and some semantic problems. This phase of the Processor is itself generated automatically by a meta-processor called a Syntax Analysis Program Generator. This phase also stores the statements in a simulated associative memory for ease in later search, analysis, and processing and prints error diagnostics in a report for the user.

In the second phase, the MODEL Processor determines precedence relationships from analysis of the specification. The independent and unordered MODEL statements are analyzed for precedence relationships which are sometimes determined by description components, and other times by assumptions based on a set of deductive or heuristic rules. These relationships are used to form a precedence graph against which the completeness and correctness of the specification can be checked, and reports are produced for the user indicating the data, assertions, or decisions that have been inadequately described and displaying the

Processor assumptions that were made in the absence of information provided by the user.

The third phase of the Processor determines the sequence of execution of all events implied by the specification, using precedence and graph theory techniques, and thereby determines the sequence and control logic of the desired module. It also deals with scope and iteration analysis. The result of this phase is a set of data structures representing the desired sequence of processes and flow of events, sequenced and ranked in their order of execution. Thus, the output of this phase is a table that is similar to a program flowchart of the desired module, and is subsequently used to produce a flowchart-like report and the program.

The fourth phase is code-generation which entails insertion of code into the entries of the flowchart to produce the program in a high-level language, PL/1, ready for compilation and execution. Code is produced in two steps for purposes of modularity and independence of the target language. The first step produces a language-independent version of the flowchart-entries, while this second step produces code in the PL/1 programming language. Code is generated for input/output commands, for procedures and their invocations, for program iterations and other control structures that are necessary, and for declarations for object program data structures and variables. A listing of the generated program as well as the flowchart-like report is produced.

Finally, the automatically produced PL/1 program module is ready for compilation by the PL/1 optimizing compiler, which carries out program compilation and optimization of code on the machine-language level. Integration of the program into the system and its subsequent execution can then take place by traditional means.

1.2 Summary of Contributions

The importance of this dissertation lies in several areas. The development of a non-procedural language for defining information systems is useful in itself to express a specification to a programmer. The formal language imposes a discipline on the specifier of the application system and allows the problem to be expressed descriptively and logically rather than procedurally and physically.

The primary result of this research is the development of a new and automated approach to producing software, an approach which offers a viable alternative to conventional manual software development, and which should reduce the cost of its development.

A major contribution of this research is the development of a series of algorithms which, taken as a whole, constitute a Processor that automatically produces programs from specifications in the MODEL language. The MODEL Processor demonstrates a structured design and programming methodology for the application system.

Regarding implementation, over 12,000 lines of PL/1 code have been written for most of the algorithms. The algorithms are given in the body of the dissertation, and the corresponding PL/1 code is in the appendix. Because the author had to assume a position in September 1975, further developmental and testing work has been left for the future; nevertheless, the feasibility of the automation of the Processor is already evident.

The MODEL Processor is an example of a knowledge-based system in that it embodies a formalized knowledge of how to write data processing programs, and thus replaces the application programmer/coder. As shown in Chapter 4, the programming philosophy formalized in the Processor does not necessarily pattern that of the typical human data-processing programmer. The Processor accepts a set of unstructured and unordered descriptions, formulae, assertions, rules, etc., and builds a structure around it, with a flow, control logic, and procedural operations not explicit in the descriptive specification.

The benefits of automating the design and generation of program modules using this approach can be seen readily in its effective utilization of manpower and machine. By imposing a discipline on the program module definer and allowing him to concentrate on the logical aspects of the program, fewer errors should result in system building. Automatic checkout of the specification by the Processor quickly and accurately pinpoints many ambiguities, incompleteness, or inconsistency within it. As confidence would be gained with the

Processor over time, the Processor could be relied on to detect problems with the specification. MODEL requires the specifier to provide data descriptions and to express arithmetic and logical relationships inherent in the problem, but relieves him from all procedural, physical, computer-oriented, sequencing, input/output, control logic, and "housekeeping" problems that are in the realm of programming. By using such a Processor, the machine can design and generate the program modules of the desired application system. Furthermore, by accepting a machine-readable specification, and by providing feedback, reports, and charts (including a flowchart-like report), the Processor also partially automates the documentation process.

Thus, the MODEL system is a step towards eliminating the application programmer and effecting a direct communication between the system specifier and the machine. It is therefore a crucial step in the automation of software development, with immediate potential for saving man-hours and cost in software development.

1.3 Organization of this Dissertation

Chapter 2 is a summary of background and motivation to this research and a survey of related literature and research. It reviews other effort and research in the automation of different aspects of software development, and is especially directed towards the reader unfamiliar with this area of research.

Chapter 3 describes the MODEL language with a more elaborate overview of its purpose and role in software development, its features, an example of its use, and a detailed description and examples of the MODEL language itself. A formal grammar of the language is also provided. Chapter 3 can be read by a user as a guide independent of the other chapters.

Chapter 4 describes the design of the MODEL Processor, including the theoretical background, system methodology, algorithms, and programming techniques.

Chapter 5 summarizes the conclusions that can be drawn based on this research, and suggests directions for further research.

An appendix is provided at the end with a sample problem expressed in MODEL and its corresponding output that would be produced by the MODEL Processor. Also included in an appendix is the source code for some of the modules of the system.

CHAPTER 2

Background, Motivation, and Survey of Related Literature and Research

2.1. Introduction

The motivation for this research arises from concerns over the rising costs of software development and the forseen difficulties in continuing to build complex software systems by the conventional manual methods. This concern is commonly reiterated in the literature of recent years as the following sample of quotations demonstrates.

Boehm [BOE 73] estimates the ratio of software to hardware costs to rise to 10 to 1 by 1985 if present trends continue and warns that

> If the software-hardware cost ratio appears lopsided now, consider what will happen in the years ahead as hardware gets cheaper and software (people) costs go up and up.

Prywes [PRY 74] concurs that

> While improvements in hardware, system software and programming languages continue to make programming more effective, they are not sufficient to compensate for the dramatic increases in demand for software... [and an increased programming] requirement would constitute not only a financial obstacle to advances in use of computers, but also such a labor force cannot be recruited or trained.

In proposing the automation of systems building, Teichroew [TEI 71] points out that

> It is extremely unlikely that it will be possible to build the number of systems of the size and complexity desired by manual methods -- there will not be enough people.

The sample of papers referenced above [TEI 71, PRY 74] and other studies [KOS 74] have recognized and proposed that a solution to the systems building task is going to have to be the utilization of the computer itself in the software systems development process. That is, there is a necessity for at least the partial automation of the software development process in order to cope with the rising software costs and shortage of sufficient trained personnel.

2.2. Improvements to the Software Development Process

Before reviewing efforts that have been made and are continuing in the automation of the various phases of the software development process, which in turn gives perspective on the research reported here, the following discussion first reviews other attempts that have been made to aid and harness the software development task.

Recent developments in software development, programming languages, and software management techniques promise to improve the productivity and quality of software to some extent. Much attention has been given in recent literature to "structured programming" [DAH 72, BAK 72b, DON 73, MIL 73], "top-down programming" [MIL 71, MIL 73], and "chief-programmer-team" effort [BAK 72a, BAK 72b, BAK 73]. These form an inter-related set of programming techniques that can shorten the software development process in the implementation phases. These include imposing a well-defined discipline on the programmer, restricting and simplifying the set of control structures to be used by programmers, modular program design from "top" to "bottom" by use of successive refinement of detail, limiting the size and complexity of modules, etc. In addition to promoting the above, writers such as Boehm [BOE 73] also suggest better management of software production to include accepted procedures such as thorough organization, setting milestones, early prototypes, setting contingency plans, making good test plans, detailed interface specifications, etc. Although these suggestions are useful if practiced and the "stuctured" developments are significant in improving software production, they are, as seen from the above comments, by no means a "cure-all" and are not likely to solve the software development problem in the long run.

The use of general application packages has been another way to reduce the cost of acquiring a software system for many years. These are mostly software packages consisting of general programs that are oriented towards a specific application area, and which are then attempted to be tailored to a specific user's needs. Application packages range in sophistication from those for which the user only enters data values, to those where he provides parameter values, to those where he selects options be means of a checklist or questionaire (such as the IBM Customizer for the System/3), to those where the user has, in addition, a command-language to select certain options. A sophisticated example of an application package that can be customized can be found in the operations management area [HAX 75]. Other examples of the latter packages are some of the auditing packages for extracting, summarizing, and formatting data [ADA 72, WIL 72].

The primary deficiency of application packages are their limitations due to lack of flexibility. The user must fit his solution into a rigid structure of the pre-written program. Furthermore, as the user requirements increase, there arises a need for special-purpose programming which requires programming skill. The requirement to have programming skill and procedural knowledge is also involved in applying the package initially and in inserting needed subroutines. Finally, some of these packages, especially those that are in the form of a library of generalized parameterized routines, tend to be inefficient in execution.

For many years, application-independent "generalized" packages have also been involved in some aspects of software from input/output subroutines, to sort/merge packages, to report generators, to file maintenance programs. All of these relieve the programmer of any application from details which are well-understood. These application-independent packages (such as general file-maintenance and report generation packages) and even those that are pre-processors to a compilation (e.g. CRAPE [GRA 72] and SCORE and others reviewed in [ADA 72]), have inherent limitations and cannot be considered to be automatic generators of programs that can meet general user requirements because they have one or more of the following deficiencies:

(1) They are tailored to a specific class of applications,

(2) They impose a rigid control structure inherent to the generated program,

(3) They require a procedural knowledge to use, or

(4) They require a knowledge of COBOL or other language for reasons such as inserting and writing subroutines.

A greater level of sophistication has been reached in recent years with the development of generalized data base management systems [COD 70, COD 71a, COD 71b, PRY 66, PRY 72]. These systems enable the user to view data in elaborate logical structures by mapping them into physical representations, relieve their users of much of the details of access and other operations, and provide for greater control and independence of data, among other services. Data base management systems, however, still require their users to have programming skill and procedural knowledge, through a conventional programming language host system, (e.g. IMS), or through an independent command language (e.g. MARK IV). They furthermore are often difficult to use, have great limitations imposed, and are often inefficient and costly in execution. In spite of these drawbacks, their current development is significant in data organization, access, management, and control, but they by no means will solve the requirement for programmers.

2.3. Research on Automation of the Systems Building Process

Only in the last few years has there been a concerted effort to automate systematically the systems building process. The ensuing discussion reviews the contribution and efforts of various projects, papers, and systems in automating one or more phases of the software development process. For discussion purposes, the typical software development process can be broken into the following steps or phases which have been alluded to elsewhere in this dissertation. These divisions are similar to those found elsewhere in the literature [COU 73, TEI 71, PRY 74]:

(1) determination of user requirements;

(2) production of system functional specifications;

(3) system physical design:

(a) evaluation and selection of files and their organization;

(b) decomposition of the system into modules;

(4) program design;

(5) coding, debugging, and testing;

(6) operations and maintenance.

The systematic automation of each of these phases has been proposed several years ago [TEI 71] in a large-scale research project, ISDOS (Information Systems Design and Optimization System), that has made progress in several areas surveyed below. The potential benefits of automating each of these stages have been subsequently evaluated [PRY 74, TEM 72]. Prywes' survey [PRY 74] estimates the percentage of cost reduction that could be effected by the automation of each of these phases, and suggests possible directions that such automation might take.

The least amount of automation progress has been made in determination of user requirements in phase(1) and the production of system functional specifications in step (2) which require knowledge of the application area and problem-solving capabilities. This has so far been automated only very minimally by severely limiting the scope of the problem domain in application-specific packages cited earlier. Such packages require manual programming of a model of the application area or "problem domain", perform a logical as well as physical design, and then generate a program based on generalized pre-stored programs.

Balzer [BAL 73] in an ARPA-sponsored project defines "automatic programming" as the process of accepting a problem in terms of a model of the domain, obtaining a solution for the problem in terms of this model, and producing an efficient computer program as the solution. Thus, Balzer's view is that of automating the entire software development process without any recognition of the division of phases above that actually require different areas and levels of knowledge. He further suggests that it will be possible to have the computer system acquire a model of the problem domain through an interactive session in a natural language. Yet success with such an approach seems to be a long way off and will not be realized until major advances are made in artificial intelligence and problem solving techniques.

Natural language processing and problem solving have so far only been successful when the knowledge base of the problem domain is so small as to be entirely representable formally (see [HEW 71, WIN 73]). As observed in [PRY 74], the possibility that the computer system itself can acquire a model of the problem domain through an interactive natural language session requires advances beyond the current state-of-the-art automatic problem-solving methods (as surveyed in [SHA 73]).

More progress has been made in the partial automation of the middle phases of software development: expressing problem statements formally, automatically analyzing system functional specifications, and producing a physical design. These are described below.

After the overall user requirements have been determined in phase (1), a step towards further automation is the expression of those requirements in a formal language. There are a number of so-called problem statement languages for expressing system functional specifications formally [TEI 72], some theoretical and some more pragmatic. The importance of such languages was recognized at least theoretically quite early with works such as those of Young and Kent [YOU 58]. The Problem Statement Language (PSL) of the ISDOS project so far seems to be the most complete specification language to express system functional specifications formally [TEI 72, HER 73]. The impact of such problem statement languages has been shown to be of great benefit [TEM 72] in aiding the software development effort by giving the problem definer a formal way of expressing the problem without having to specify how the task will be carried out and by putting such specifications into a form that they could be analyzed automatically.

Once the functional specifications have been expressed formally in machine-readable form, a further degree of automation has resulted by automatically checking the specifications and producing some of the systen phyical design. The ISDOS project [TEI 71] and Nunamaker [NUN 71] to name two, have made inroads into the automatic analysis of problem statement languages and the automatic physical design and optimization of information systems. The ISDOS project under the direction of Professor Teichroew [TEI 71, TEM 72, HER 73] has developed a system which so far performs analysis of functional specifications expressed in a Problem Statement Language at the system level and generates numerous reports about the specification. Nunamaker's work [NUN 69, NUN 71, NUN 72a, NUN 72b] has led to a system which can perform the file and module design of an application system with sequential files by evaluating the different feasible designs and selecting the "best". A more general automatic file and module design system has been proposed and is currently being pursued by Gibb [GIB 75].

More limited aids to the systems analyst have actually been available previously in forms-oriented and tabular languages (see for example [ADS 68, TAG 68]). In fact, a machine-readable and machine-checked version of the forms-oriented Accurately Defined Systems (called PSL/ADS) had also been implemented as part of the ISDOS project [THA 71] as a predecessor to the current more complete PSL/II

[HER 73].

Other efforts on automatic physical systems design (phase 3 above) have resulted in a model for selection of file organizations for a simple system [SEV 72] and in a system for the automatic evaluation, simulation, and selection from alternative file organizations [CAR 73]. Numerous other papers have been written which shed light on the automation of the physical design process through formal models and structured systems development techniques (see, for example, [PAR 72, SHE 74, GRA 73]).

The contributions of the above systems and research papers have been primarily in partially automating stages (2) and (3) above. These systems as yet are not involved in automating the latter stages; i.e. they do not generate programs based on previous non-procedural specification, analysis, and design. These efforts have enabled automatic checking of system functional specifications and production of some of the system physical design based on analytic grouping decisions or simulation techniques.

The research reported in this dissertation helps to bridge the automation gap by attacking the automation of phases 4 and 5 above -- the program design, coding, and debugging phase. As explained in detail in Chapter 3, MODEL is a language for describing modules of an information processing system which has previously gone through phases 1 through 3, the logical and physical design process, and an important feature is its non-procedural nature. Non-proceduralness (discussed in greater detail in Chapter 3) has been recognized as an important feature for many reasons [YOU 65, TEI 72, PRO 74, PRY 74], such as enabling descriptive statements that can appear in arbitrary order, relieving the user from sequential and detailed steps, etc. The MODEL language has facilities to describe the module source and target files, data descriptions, and assertions to provide data interrelationships. The data description portion of the language is similar to data description languages [COD 71b, PRY 72, SMI 71, SIB 73, MEF 74], and in fact is an extension and outgrowth of a data description and translation project of which the author was a member [RAM 73, RAM 74, RIN 74].

The MODEL Processor, discussed in Chapter 4, has the task of accepting MODEL specifications, analyzing and checking them, and producing executable programs.

There are other efforts currently on-going in the automation of the program generation phase of software development, but which vary in orientation and approach. Nunamaker (in a continuation of work begun in the ISDOS project [BLO 73]) is currently pursuing work on automatic generation of transaction-oriented programs for Data Base Task Group-type data base management systems environments, though the language seems to require procedural supplements on the part of the user. The MODEL approach has a greater degree of non-proceduralness and generates a traditional but efficient program using the operating system's access methods.

The automatic program design and coding phase is apparently also encompassed in the automatic programming project of the ARPA sponsored Automatic Programming Group at the Massachusetts Institute of Technology. Other than ISDOS, this is another major research project whose eventual aim is the total automation of all phases of the software development process outlined above, and it incorporates both application-dependent knowledge with user interaction in natural language and computer-dependent knowledge. From preliminary indications [MAR 74, RUT 74], their automatic programming system seems to require procedural knowledge, is more theoretical, and is not as user-oriented or general as the PSL/PSA of ISDOS and other non-procedural approaches such as the research reported in this dissertation.

Another automatic program generation project is being pursued at the University of Pennsylvania in the area of automatic testing systems [PRY 75]. Some of the code written by this author for MODEL is being used in the implementation of that project.

From the above survey, it is clear that there is not yet one complete system that automates the entire software development process by taking system-level functional specifications through physical design and program generation, although there have been inroads into many of the steps. The research reported in this dissertation is intended to supplement the above group in its contribution towards the automation of the program design and implementation process.

CHAPTER 3

The MODEL Language

3.1 Introduction

3.1.1 Overview of Purpose and Usage

One of the initial goals of this research was the development of the Module Description Language (MODEL), a very high level non-procedural language in which a business systems analyst could describe a desired application information system.

Specifications in the MODEL language would be submitted to the MODEL Processor which would for the most part automate the program design and implementation process.

Before describing the nature of the MODEL language and its "non-procedural" aspects, this section first turns to its purpose and role. In order to understand the role of the MODEL language within the conventional software development process, the stages of software development are defined below. There is really no fine line between phases, and the phases do overlap. Similar divisions of the software development process can be found in other literature [COU73,PRY74,TEI71]. The following delineation of the software development process refers to Figure 3.1 (taken from [PRY74]). A discussion of which phases of the software development process the MODEL language and Processor automate is presented after the following breakdown.

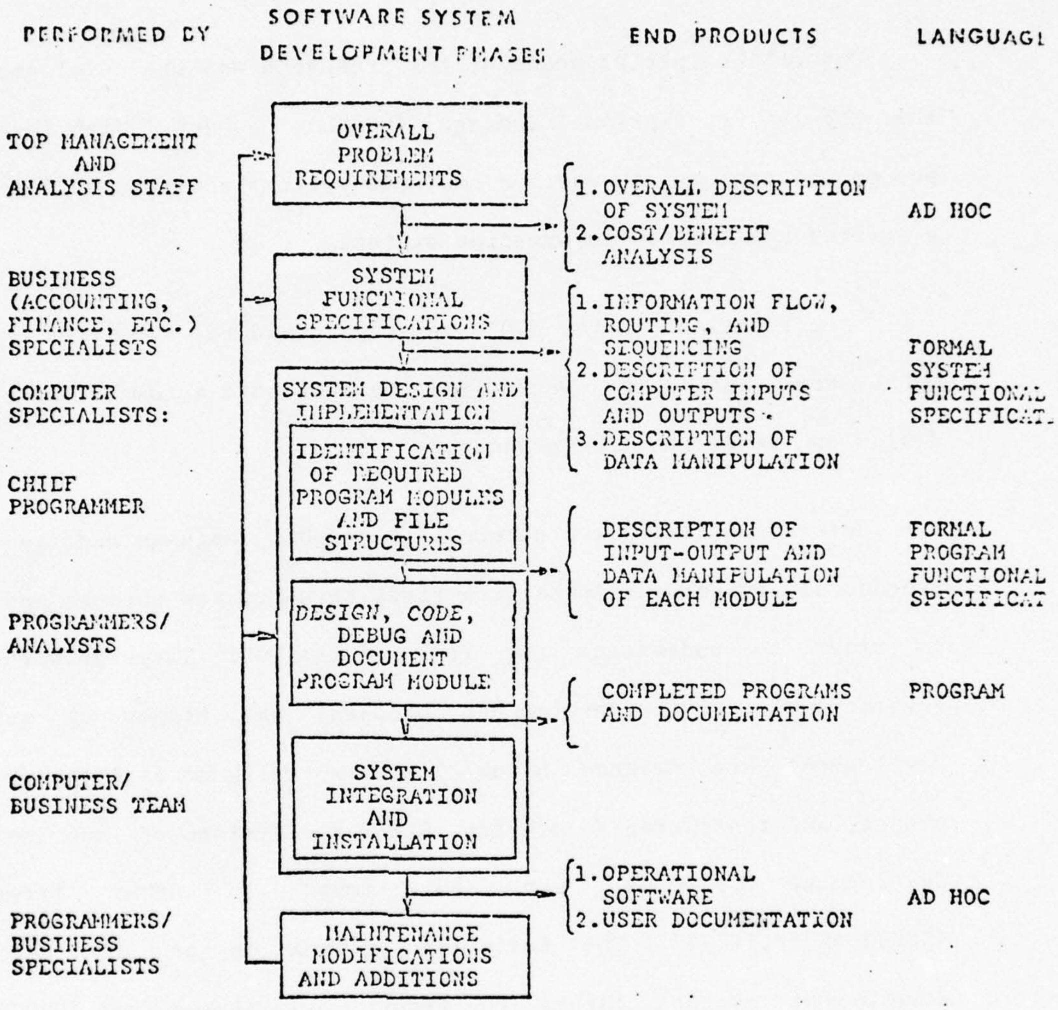

PERFORMED BY
SOFTWARE SYSTEM DEVELOPMENT PHASES
END PRODUCTS
LANGUAGE
TOP MANAGEMENT AND ANALYSIS STAFF
OVERALL PROBLEM REQUIREMENTS
1. OVERALL DESCRIPTION OF SYSTEM
2. COST/BENEFIT ANALYSIS
AD HOC
BUSINESS (ACCOUNTING, FINANCE, ETC.) SPECIALISTS
SYSTEM FUNCTIONAL SPECIFICATIONS
1. INFORMATION FLOW, ROUTING, AND SEQUENCING
2. DESCRIPTION OF COMPUTER INPUTS AND OUTPUTS
3. DESCRIPTION OF DATA MANIPULATION
FORMAL SYSTEM FUNCTIONAL SPECIFICAT
COMPUTER SPECIALISTS:
SYSTEM DESIGN AND IMPLEMENTATION
CHIEF PROGRAMMER
IDENTIFICATION OF REQUIRED PROGRAM MODULES AND FILE STRUCTURES
DESCRIPTION OF INPUT-OUTPUT AND DATA MANIPULATION OF EACH MODULE
FORMAL PROGRAM FUNCTIONAL SPECIFICAT
PROGRAMMERS/ ANALYSTS
DESIGN, CODE, DEBUG AND DOCUMENT PROGRAM MODULE
COMPLETED PROGRAMS AND DOCUMENTATION
PROGRAM
COMPUTER/ BUSINESS TEAM
SYSTEM INTEGRATION AND INSTALLATION
1. OPERATIONAL SOFTWARE
2. USER DOCUMENTATION
AD HOC
PROGRAMMERS/ BUSINESS SPECIALISTS
MAINTENANCE MODIFICATIONS AND ADDITIONS

FIGURE 3.1

STAGES IN A SOFTWARE DEVELOPMENT PROJECT

(1) Determination and Statement of Overall Problem Requirements

The initial phase of a large software development project is concerned with the information needs of management. This phase is prompted by top management and involves personnel with expertise in the application area. The result of this phase would be an overall description of the information system and a preliminary cost/benefit analysis.

(2) Analysis of Problem Requirements and Production of System Functional Specifications

This phase involves analyzing the overall system description. It consists of describing the overall information flow involving personnel, communications, manual processes, and computers, and defining transactions, documents, reports, and other information in the system. In this step, the computer is viewed as one big black box. The functional specifications produced in this stage describe the information and transactions that go in one end of the system and all the reports and information that come out of the system. In a well-managed software development project, the specifications will be as formal, precise, and complete as possible.

(3) System Physical Design

The system design phase consists of several activities. One is decomposing the desired system into logical sub-units or modules. Another is organizing and designing the aggregate of data mentioned in the functional specifications into files, based on such factors as frequency of use, location of use, efficiency considerations, etc. A third activity is selecting the files in and out of each module. This phase is performed by computer specialists, including analysts and senior programmers. Each identified program module is typically associated with a transaction, an updating function, or a reporting function. The products of this phase are the file structure specifications, record layouts, system flowcharts, and the program module logical specifications which incorporate the system specifications on a per-module basis.

(4) Program Design

The program design phase is concerned with the detailed logic of each module of the system. It involves analysis of the input and output data of the module and analysis of the specification of information relationships and algorithms to be used within the module. From this knowledge, this phase has to enumerate and sequence the events to take place in the module; i.e. the design of the sequence and control logic. This has traditionally been written in the form of a program flowchart by a programmer or programmer/analyst.

(5) Coding, Debugging, and Testing

This is the final implementation phase where the data is declared, the processes are ordered, and input/output statements are included. All this is embedded in appropriate control structures, and coded by programmers in a compiler language such as COBOL or PL/1. Debugging and testing proceed to check out the program.

The term "program module" in this entire discussion refers to a logical sub-unit of a larger information system, such as the "sale module" or "reorder module" of a department store inventory system. This looser use of the term "module" may not necessarily correspond to that in literature on top-down and structured programming [MIL71,BAK72b,BAK73] use of the term, where it is rigidly urged to keep "modules" or sub-programs less than a specified length (usually about one page). Thus a module corresponding to a logical function or component of the system may have sub-modules.

Since the immediate goal of this research is the automation of stages 4 and 5 above, the program design and coding phases, the MODEL language enables a business system analyst to describe each desired program module formally, after he completes the stages above it. In other words, MODEL would be used to formalize a design after the statement and analysis of requirements is complete, after the information system is designed and decomposed into modules, and after the files in and out of each module are defined. The MODEL user would then write a formal MODEL specification for each of the modules in the system and submit these descriptions to the MODEL Processor. In turn, the Processor would automatically complete the program design phase and proceed to generate the program, thus replacing the coding, debugging, and testing phases.

The analysis and design phases preceding the use of MODEL will be done by the conventional manual method, at least for the near future. It is envisioned, however, that these phases, too, will some day be at least partially automated and that specifications in a language such as MODEL could be produced automatically, from a formal functional specification language. Several such so-called "Problem Statement Languages" exist in the research community [HER73,TEI72], and it would be possible, for example, to take PSL of the ISDOS Project [HER73,TEI71] and transform it into MODEL-like statements by designing the files, decomposing the system into modules, etc. Research on automatic evaluation and selection of file organizations has been reported [CAR73] as well as work on automatic system design [NUN71,PAR74]. Other research that would enable such a totally automated system is actually in progress [NUN71,GIB75,TEI71].

Note that in Figure 3.1 the MODEL Processor automates the phase of software development that is labelled "design, code, debug, and document program modules", which corresponds to phases 4 and 5 above. All the boxes above it would have to precede the use of MODEL.

3.1.2 MODEL Language Components and Novel Features

A specification of a module in the MODEL language consists of a set of descriptive statements about the desired module. It describes:

(1) which files go in and out of the module; i.e. which files are source and target of the module);

(2) a description of each file, its structure, components, layout, and inter-relationships (to be described further);

(3) selection rules defining subsets of data to be considered; and

(4) assertions on data relationships (to be described further).

Section 3.2 will give a detailed description of the MODEL statements including formal definitions and examples. Some of the novel features that were incorporated into the language design are summarized here as follows:

(1) Non-proceduralness

Much attention has been given in recent literature [PRO74] to non-proceduralness in languages. Leavenworth and Sammet [LEA74] stated that non-proceduralness is a relative term and that some languages are less procedural than others. Although they give no metric for measuring the degree of non-proceduralness in a language, they do give a list of characteristics that a language needs if it is to be considered non-procedural or purports to lower the level of proceduralness. Their list of characteristics includes minimization of the amount of sequencing of statements required by the user; a capability to state a problem in terms of structures relevant to the problem rather than in those operations convenient for some machine organization; aggregate operators (e. g. the FOREACH feature in MODEL); associative referencing (whereby the user does not have to specify explicit access paths or write an algorithm to search a data structure). By their criteria, MODEL would certainly possess a high degree of non-proceduralness. Specific aspects of non-proceduralness incorporated into MODEL are the following:

(a) declarative or descriptive statements as opposed to imperative (command) statements: each statement describes data or data relationships to other data not commands.

(b) statement order independence: each statement consists of an independent description, and the statements can be submitted in any order.

(c) specification modularity and statement independence: the statements can also be added in independent stages; this corresponds to the notion of incrementality found in another non-procedural language dealing with automatic test equipment [PRY75].

(c) absence of control structures: since the statements are indpendent descriptions and can go in any order, there are no control structures connecting them. The sequencing and control logic code will be produced automatically by the Processor;

(d) absence of computer programming terminology and concepts: namely, no reference to sequences of operations, control code, input and output commands, counting, memory, computer implementation, or generally any processes.

(2) Machine Independence

The MODEL language describes an application module without reference to a particular operating environment or computer system, and like high-level languages, it is intended to be usable on a variety of machines. Automatic generation of the application programs by the Processor in a high-level compiler language, namely PL/1, also promotes machine independence and transferability.

(3) Domain/Application Independence

The MODEL language is concerned with describing information and information relationships. The MODEL Processor has only computer programming knowledge. Thus it should be applicable to a broad spectrum of applications.

(4) Capability to Store, Control, and Share Data Descriptions

The MODEL language was designed with features in mind that in the future would enable sharing common data descriptions in different modules and allow a data description library to be maintained centrally. This would facilitate the use of MODEL in a shared data base environment and promote data and program independence as shown below.

(5) Data and Program Independence

The capability to store and share data descriptions promotes a capability to monitor changes made to the data description files. Thus, whenever changes are made to the data description files, they could be accessed by all MODEL specifications using them. It would then be possible to invoke the MODEL Processor in order to have it automatically regenerate affected modules. Therefore, the efficient method of compile-time binding of the data description to the program still can be maintained, while having program and data independence, by automatic regeneration of affected modules.

This last concept is depicted in Figure 3.2. It is proposed here that the MODEL Processor be augmented with an automatic monitor to record changes to the data base description made via an editor and with an index correlating every data description set with the modules that utilize it. In this way, the MODEL Processor could be invoked automatically by the monitor to regenerate the affected modules when

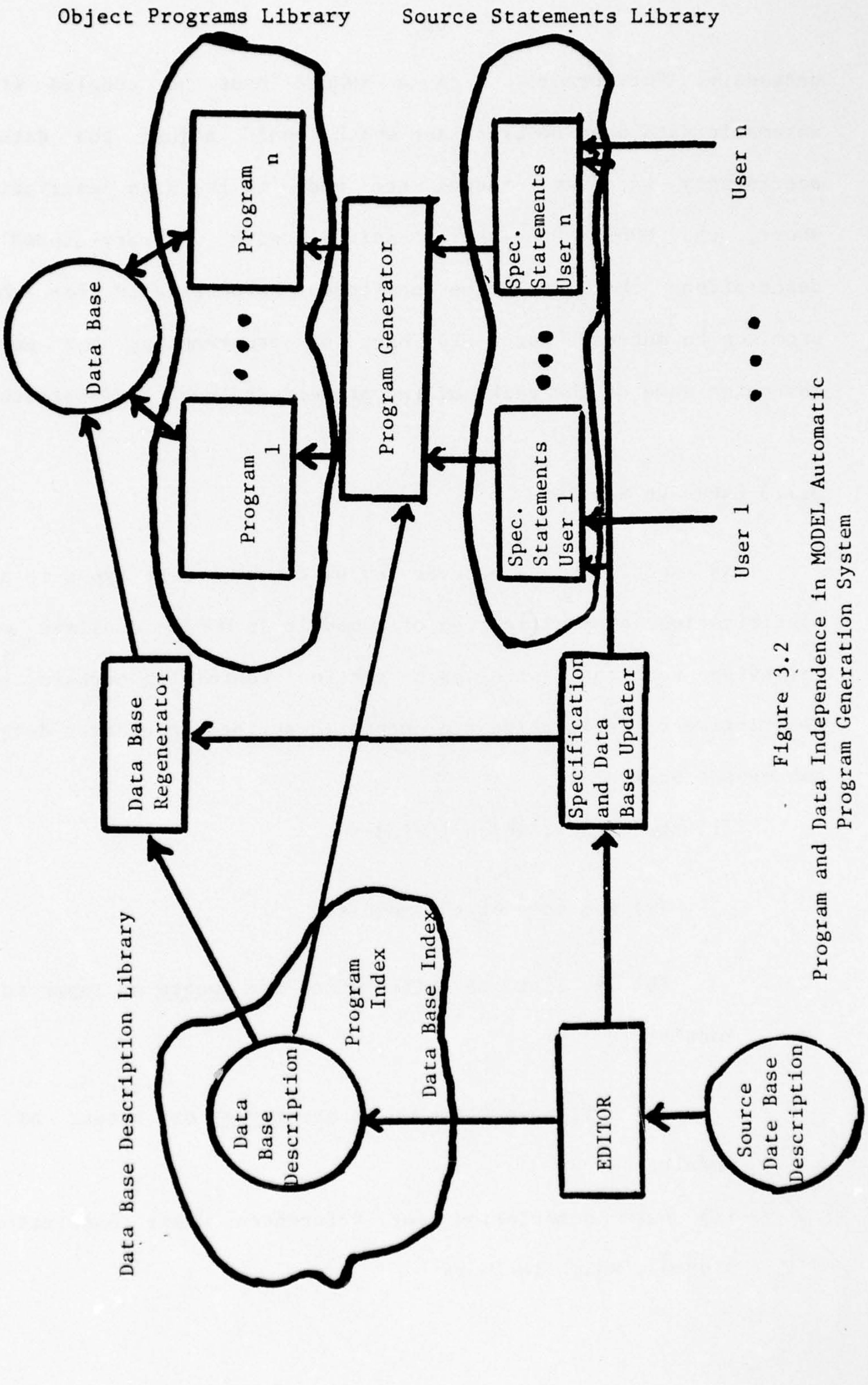


Figure 3.2

Program and Data Independence in MODEL Automatic Program Generation System necessary. Furthermore, such a scheme must be coupled with an automatic data base re-organizer which would adjust the data base accordingly whenever changes are made to the data description. In short, the MODEL Processor, paired with library-stored data descriptions that could be monitored automatically for changes, promises to automate not only part of programming, but partially automates some of the tasks of the present-day data administrator.

3.1.3 Language Sections

This section gives an overview of the statement types in a MODEL specification. A specification of a module in MODEL consists of the following sections, with each section containing certain types of descriptive or declarative statements (described in greater detail in subsequent sections).

(1) the header, which includes

(a) the name of the module;

(b) a list of files which are source or input to the module;

(c) a list of files which are target or output of the module;

(2) data description (or references where description is stored), which includes

(a) statements describing each file, its component records, groups, and fields, the medium on which it is stored, and statements classified as assertions. These define the length of variable-length fields or the number of occurrences of variably-existing groups and fields. Alternatively, the data can be described by referring to a stored description in a library;

(b) inter-file relationships;

(3) Module-Specific descriptions, which include

(a) descriptive statements of any interim variables not found in either source or target file descriptions;

(b) source-set assertions which describe the subset of source data to be considered;

(c) target-set assertions which describe the subset of target data to be considered;

(d) other assertions defining information relationships, data computations, or decision rules.

Figure 3.3 summarizes the components of a MODEL specification. It presents an outline of the methodology for a MODEL user to employ in preparing his specifications; namely that the following steps be taken:

HEADER

- Module Name
- Source Files
- Target Files

DATA DESCRIPTION

- File, Record, Group, Field Description
- Media Description
- Inter-file Relationships

MODULE-ORIENTED STATEMENTS

- Interim Variable Descriptions
- Selection Criteria
- Assertions for Information Relationships, Computations, and Decision Rules

Figure 3.3

Components of a MODEL Specification

(1) the module should be named;

(2) the source files should be listed;

(3) the target files should be listed;

(4) each file should be described, including a description of the file structure, the storage medium, record layout, and its sub-components; namely, a description of each field and its attributes. Furthermore, statements (called LEN and EXIST assertions yet to be described) should be provided for each variable-occurring or variable-length field respectively; such statements are used dynamically to evaluate such data-dependent quantities;

(5) statements providing inter-file relationships should be included to supplement the file descriptions with relationships between files;

(6) interim variables which are neither in source files nor in target files should be described;

(7) source set assertions providing selection criteria for source files should be given;

(8) target set assertions providing criteria for target files should be given; and

(9) assertions describing information relationships, data computations, and decision rules should be provided.

Section 3.2 will present detailed rules, methodology, and examples for writing the statements of a MODEL specification, but first an example of an application of MODEL is given for the reader to become familiar with the language.

3.1.4 An Example of an Application of MODEL: The DEPSALE Problem

This section describes a class of data processing applications for which MODEL is well-suited, and gives an example of the use of MODEL by way of a specific problem. From the previous sections, it can be inferred that the MODEL language can be used to describe desired application programs in many areas of data processing. The class of programs that MODEL is capable of generating has already been discussed in Chapter 1. MODEL can be employed in applications where large amounts of source data come in from already designed and existing files; data gets processed, transformed, or computed upon according to certain defined rules; and target data is produced. A current restriction is that files must be of either sequential or indexed sequential organization.

In order to make it easier to discuss the various aspects of the MODEL language and exemplify its usage, a sample problem called the department store sale problem (DEPSALE) is provided and defined in MODEL. Its complete specification in the MODEL language, and its listings and reports can be found in Appendix A. References are made to the DEPSALE problem throughout the remainder of this chapter to exemplify various statements in MODEL .

The example deals with a department store whose customers normally purchase items from stock. As seen in Figure 3.4, the function to be specified maintains customer, inventory, and sale data and issues invoices for the purchasers. A MODEL description of this requirement consists of describing source and target data and business decision rules. The specifier need not make any references to processes, computers, or events. Obviously, the computer process is implicit in that one purpose of the description would be to produce a program that will generate the target data automatically.

Assume that the specifier is a department store analyst expert in management and accounting practices, but has no computer oriented training. He first describes the source and target data in MODEL; that is, the information obtained from the purchaser (entered into the point of sale terminal), the composition of a customer-master file, a stock inventory file, a sales journal, and a customer invoice.

In addition, the analyst specifies rules for decision, accounting, conversion, and other rules (called assertions in MODEL) that would indicate dispositions in certain eventualities, such as when a stock item is not available from inventory, or when a purchaser exceeds the allowed credit limit. The accounting rules specify the determination of charges for purchases and the method of determining the customer's balance.

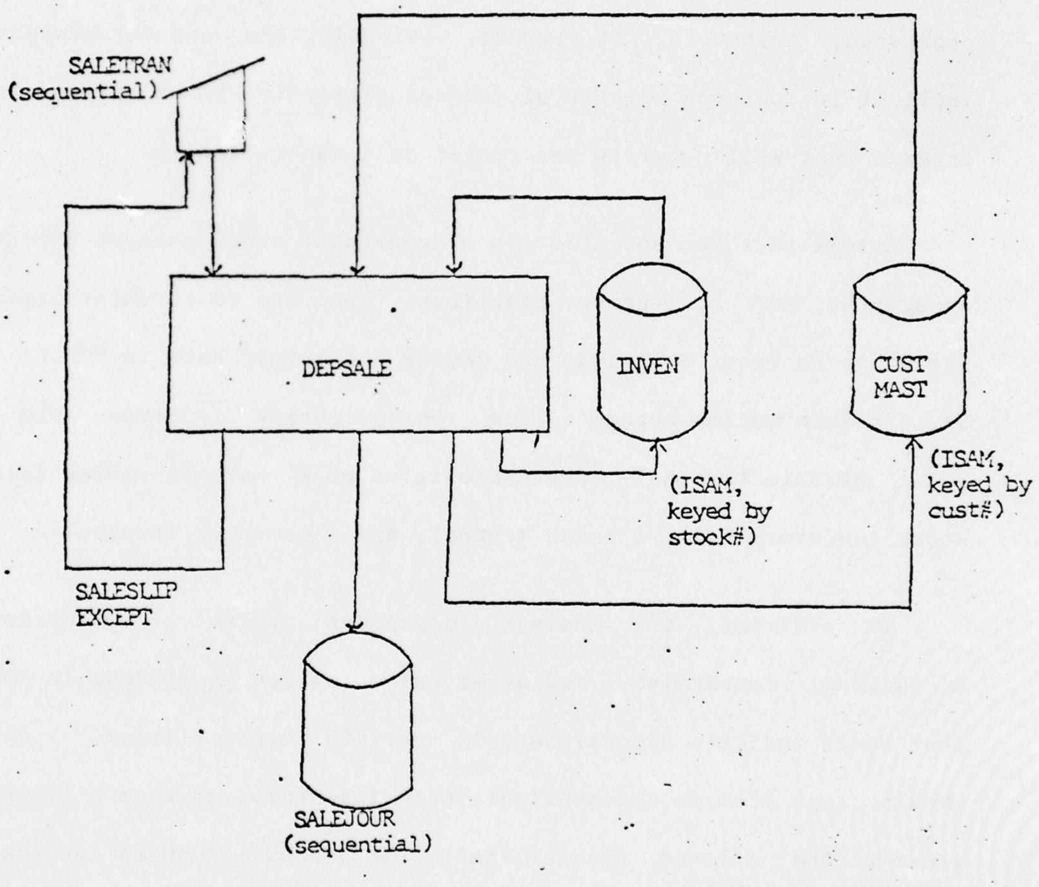


FIG. 3.4 ILLUSTRATION OF DEPARTMENT STORE SALE (DEPSALE)

In the above cited example, the department store analyst does not need computer programming knowledge since the references to a process or computer system is only implicit. Also, although he uses his knowledge of the department store business and department store vocabulary to provide data descriptions and assertions, the vocabulary that is inherent to MODEL does not require specific application knowledge. Therefore, an analyst in another business application, such as logistics or insurance, could utilize MODEL as well, and give the MODEL specification the flavor of his own application. More specifically, the vocabulary of words inherent to MODEL is oriented towards data and assertion description. For his own convenience, the user can name the data, records, files, and assertions using words meaningful to his own particular applications.

In the future, MODEL program module specifications could include descriptions and assertions of data which may be common to a number of programs. Thus once a user has described data or assertions, the descriptions could be stored in a library and there would be no need to re-describe when specifying another module that shares the same data or assertions.

3.2 User Reference Manual for MODEL

This section serves as a reference for using the MODEL language. It describes in detail the various statements MODEL. For each statement, its purpose, syntax, and semantics are given, along with an example of its use, usually from the DEPSALE problem.

3.2.1 General Information

A specification of a module in MODEL consists of a set of statements which describe the desired module. Although these statements can appear in any order, the specification can be divided into the following sections for purposes of organization.

(1) a header whose statements describe the overall source and target files to the module;

(2) a data description section, whose statements describe the media, files, records, groups, and fields;

(3) an assertions section in which assertions are stated for data relationships and other purposes to be described.

The description of each of the statements starts with sub-section 3.2.2, following a few general notes here.

3.2.1.1 Syntax Notation

In the descriptions of the syntax of MODEL statements below, upper case letters refer to specific MODEL vocabulary or to user provided names and angle-bracketed < > lower case letters refer to a generic class for which a specific item needs to be substituted. The symbol "::=" means "is defined as". Square brackets [ ] indicate optional portions of the statements. Asterisks following square brackets [ ]* signify repetition zero or more times. The "|" symbol means "or", and is used for alternatives.

The formal syntax of MODEL is provided in Section 3.3 in the Extended Backus Normal Form (EBNF) specification language as a supplement.

3.2.1.2 Names

Whenever a <name> is indicated in the statements of MODEL, it may consist of 1 to 8 characters defined below (except file names which are limited to 6 characters due to various technical restrictions). The names chosen by the user, however, should not be one of the set of reserved words: ANSI_STD, ASSERTIONS, BIN, BINARY, BLOCKSIZE, BYPASS, CARD, CHAR, CHARACTER, CHOICE, CYL, DELIM, DENSITY, DISK, EVEN, FIELD, FILE, FILES, FIXED, FOREACH, FUNCTION, GROUP, IBM_STD, INDEXED, INDEXED_SEQUENTIAL, INTERIM, INT_NAME, IS, ISAM, KEY, MAX_BLOCKSIZE, MAX_RECORDSIZE, MODULE, NONE, NO_TRACKS, NUMERIC, ODD, ORG, ORGANIZATION, PARITY, PRINTER, PUNCH, RECORD, RECORD_SIZE, REPLACE, REPORT, REPORT_ENTRY, SERIAL# SEQUENCE, SPACE, SOURCE, START_FILE, STORAGE, SUM, TAPE, TAPE_LABEL, TARGET, TERMINAL, TODAY, TRACKS, UNDEFINED, UNIT, UNITS, VARIABLE, VARIABLE_SPANNED, VOL_NAME.

3.2.1.3 Character Set

The characters which can be used to form names can be any combination of 1 to 8 letters, digits, or special characters defined below, but the first character of a name must be a letter.

<LETTER>::=A | B | ... | X | Y | Z

<DIGIT>::=0 | 1 | 2 | ... | 9

<SPECIAL-CHARACTER>::= @ | _ | #

3.2.1.4 Integers

Whenever an <integer> is indicated, it may be any combination of digits with values from 1 to 32767, except where further limits are indicated.

3.2.2 Specification Header

The header of a MODEL Specification consists of three statements, the module name statement, the source files statement, and the target files statement, described below. Taken together, they form the equivalent of the block diagram of the desired module.

3.2.2.1 Module Name Statement

Purpose: to give a name to the desired module.

Syntax:

```
MODULE: <name>;
```

Semantics: <name> is given to the module.

Example:

```
MODULE: DEPSALE;
```

3.2.2.2 Source Files Statement

Purpose: to indicate the names of those files which are sources or inputs to the desired module.

Syntax:

```
SOURCE [FILES]: <name> [,<name>] * ;
```

Semantics: The list of named files are source files to the module.

Example: in order to indicate that the files named TRANS, INVEN, CUSTMAST are source files to the DEPSALE module:

SOURCE FILES: TRANS, INVEN, CUSTMAST;

3.2.2.3 Target Files Statement

Purpose: to indicate the names of those files which are targets or outputs of the desired module.

Syntax:

```
TARGET [FILES]: <name> [,<name>]*;
```

Semantics: The list of named files are target files of the module.

Example: in order to indicate that the files named SALESLIP, JOURN, EXCEPT, CUSTMAST, and INVEN are target files of the DEPSALE module:

```
TARGET FILES: SALESLIP, INVEN, JOURN, EXCEPT, CUSTMAST;
```

Notice that a file can be both source and target such as a file to be updated (e.g. INVEN).

3.2.3 Data Description

The data description statements* describe each of the files or reports in the user's desired module and their component records, groups, fields, and associated assertions. It is envisioned that in the future these descriptions could come from a library of many file descriptions. Each file is described both logically (via the FILE, RECORD, GROUP, FIELD statements) and physically (via the STORAGE statements). Each component of the file is described in a separate statement.

The data description statements are purely descriptive in nature. They enable the user to describe source and target files that are sequential or indexed in organization, that are ordered or unordered, and that can appear on a variety of storage media used in today's data processing community.

---

* The data description portion of MODEL is a modification and expansion of the DDL language of the University of Pennsylvania's DDL Project on which the present author was an active participant. Documentation of that language can be found in [RAM73] and [RIN75]. Many modifications were made to it by this author for MODEL including the following:
the DDL within MODEL consists of only descriptive statements. The CONVERT, SCAN and other conversion commands were removed; the data mapping commands were replaced by the assertion section of MODEL and new implicit rules. All references to the old conversion process in DDL, including WRAPUP, PRECRIT, POSTCRIT, CONV commands, were replaced in MODEL by non-procedural statements. The DDL for MODEL was enhanced with facilities to enable descriptions of new file organizations (indexed); descriptions of key fields; dynamic non-procedural expressions of varying-lengths and repetitions; other physical storage media; inter-file relationships; descriptions of multiple independent files, etc.

AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416

UNCLASSIFIED 76-05 NL

2 OF 6
AD
A032638

In descriptions of hierarchical intra-record structures, the data description capabilities of MODEL are roughly equivalent to at least those of COBOL. In addition, MODEL provides facilities to describe other attributes such as varying-length fields and variably-repeating and optionally-existing groups or fields.

The following sub-sections explain and exemplify the data description statements of MODEL, which the user would employ to define the structure and attributes of each file and its components involved in the desired module.

3.2.3.1 File Statement

Purpose: to describe a file and some of its attributes.

Syntax:

```
<file name> IS FILE(RECORD IS <recname>
[,]  [STORAGE  IS  <storage name>]
[,]  [{ KEY  |  SEQUENCE } IS <key name>] )
```

Semantics: <filename> is the name of the file (recall that file names are limited to 6 characters, due to technical restrictions whereas all other names can be up to 8 characters); <recname> is the name of the proto-type record within the file. The <storage name> in the optional STORAGE clause is used to give the name of the statement which will describe the physical attributes of the device on which the file is stored. The default device for input files would be a card reader and for output files, a printer. The <key name> in the optional SEQUENCE

IS or KEY IS clause is used to indicate on which field the file is ordered. The record, storage medium, and key field named above need to be described further in a record statement, storage statement, and field statement, respectively.

In all the above discussion, the word 'REPORT' may be substituted for 'FILE', and the word 'REPORT_ENTRY' may be substituted for 'RECORD', in order to describe a report rather than a file. In the current version of MODEL, no further provisions are made for report description, and reports shall be treated as files. This decision was made knowing that report description and generating facilities are known technology, and such report facilities as headings, footings, tabs, etc. could be incorporated into MODEL.

Example:

```
INVEN  IS  FILE (RECORD IS INVREC, STORAGE IS INVDISK, KEY IS
STOCK#);
```

which says that INVEN is a file whose records are named INVREC. There will be another data description statement defining the structure of INVREC and so on. Furthermore, the statement says the file is stored on a device named INVDISK also to be described in another data description statement, and that the file is ordered or keyed by a field named STOCK# to be described later.

3.2.3.2 Storage Statement

Purpose: to describe the medium on which a file is stored, and the physical attributes and organization of the file.

Syntax:

```
                +
                | CARD
                | PRINTER [(LINE_LENGTH =<integer>) ]
  <name> is <     TAPE (<tape-description>)
                | DISK (<disk-description>)
                | TERMINAL (<terminal-description>)
                +

  <tape-description>::=
     <record-format> ,VOL_NAME =<name>
        [,INT_NAME= <name>]
        [,NO_TRKS= <no-trks>]
        [,PARITY= <parity>]
        [,DENSITY= <density>]
        [,TAPE_LABEL= <tape-label>]
        [,START_FILE= <integer>]

     <no-trks>::=7 | 9
     <parity>::= ODD | EVEN
      <density>::=200 | 556 | 800 | 1600
     <tape-label>::= IBM_STD,INT_NAME= <name>
        |ANSI_STD,INT_NAME= <name>
        | NONE
        | BYPASS

     <record-format>::= FIXED , BLOCKSIZE= <integer>
        [,RECORDSIZE= <integer>]
       |VARIABLE , MAX_BLOCKSIZE= <integer>
        [, MAX_RECORDSIZE= <integer>]
       |VARIABLE_SPANNED , MAX_BLOCKSIZE= <integer>
        [,MAX_RECORDSIZE= <integer>]
       |UNDEFINED , MAX_BLOCKSIZE= <integer>


  <disc-description>::=
     [ORG= <org-type>]
      <record-format>
        [,VOL_NAME= <name>]
        [,INT_NAME= <name>]
        [,UNIT= <type-disk>]
        [,SPACE= <units>,<integer>[,<integer>]]
```

```
<org-type>::=   SEQUENTIAL   |   ISAM   |   INDEXED   |
     INDEXED_SEQUENTIAL
<type-disk>::=2314 | 2311 | 3330 | 2305
<units>::= TRACKS|CYL| <integer>
```

Semantics: <name> is the name of the storage medium as given in the STORAGE clause in the corresponding FILE description statement. The storage medium is any in the above list, each with its corresponding physical characteristics of the file.*

The CARD and PUNCH types need no further attributes.

The physical attributes for the TAPE medium require the information indicated, including the record format (which can be fixed, variable, or undefined),record size, and block size. The tape label alternatives have the following meaning: IBM standard or ANSI standard which require the internal name; BYPASS means that tape label processing is to be bypassed; NONE denotes no tape label. The alternatives for tracks, parity, density, and file number within the tape should be self explanatory.

---

* It should be explained that the record length, and much of this physical information, could be (and should be) produced by the Processor automatically including automatic generation of JCL. Such features were put into the language as a temporary compromise for expediency.

The physical attributes for the DISK medium needs the information indicated for the organization (sequential or ISAM, sequential being the default), record format (fixed, variable, variable spanned, or undefined) record size, block size, volume name, internal name, unit type, and space. The units for the space information can be tracks, cylinders, or an integer which indicates number of bytes. The next integer in the space description indicates the number of units of primary allocation while the last integer indicates secondary allocation amount.

The physical attributes for the TERMINAL medium have a similar description.

The device specification facilities in MODEL represent an apparent contradiction to the non-procedural and non-physical approach inherent in MODEL. The reason for their incorporation is to facilitate the use of MODEL in modification or additions to existing systems, where the hardware environment is determined by existing facilities. Generally, however, such information on the part of the user presupposes that the file design and selection have already been performed. The analysis leading to choice of source and target data media is to be performed either manually by the conventional manner, or some day, automatically from a Problem Statement Language.

Example:

INVDISK IS DISK (ORGANIZATION=ISAM, VARIABLE, MAX_BLOCKSIZE=3700, MAX_RECORDSIZE=37, VOL_NAME=INVOL, UNIT=2314); (see footnote on page 55)

which should be self-explanatory.

3.2.3.3 Record Description

Purpose: to describe the record and to name its first-level components in the tree-structured record.

Syntax:

```
<record name> IS RECORD ( <item> [,<item>]* )
<item>::=<item name> [(<minimum> [:<maximum>])]
```

Semantics: <record name> is the name of the record as was given in the RECORD clause of the FILE description statement. Each <item> is a group or field which is a component of the record in the first-level of a tree-structured record. Each <item name> is the name of a component group or field. <minimum> is an optional integer indicating that the group or field repeats that many times. A <maximum> integer is optionally used in addition if the field or group repeats a variable number of times. If both a minimum and maximum appear, then the following must hold: 0 <= <minimum> < <maximum> < 32768. The attributes of each member item are described further in later GROUP or FIELD description statements.

When a field or group is described to occur a variable number of times, the user is also required to provide an EXIST type assertion which computes the number of occurrences (see sub-section 3.2.4.2 dealing with such assertions).

Example:

STUDENT IS RECORD (NAME, SS#, DEPT(2), #COURSES, COURSES(1:6))

where STUDENT is the name of the record, and NAME, SS#, DEPT#, #COURSES are the component fields or groups to be described further in later statements. Notice that DEPT occurs twice while COURSES repeats a minimum of one time and a maximum of 6 times.

An example from the DEPSALE problem is the following:

INVREC IS RECORD (STOCK#, ITEMDESC, SALPRICE, QOH, TAXCODE, SUBST# (0:5));

Since SUBST here repeats a variable number of times, an EXIST type assertion would also be needed (see sub-section 3.2.4.2 dealing with such assertions).

3.2.3.4 Group Description

Purpose: to identify the name of a group and its sub-components, each of which in turn can be a group or a field.

Syntax:

```
<name> IS GROUP ( <item> [,<item>]* )
```

Semantics: the group name itself should have been a component member within another group or in the record in a RECORD or GROUP description statement. The <item>s have the same meaning as in the record description statement. If an item (group or field) is defined to repeat a variable number of times, an EXIST type assertion is also needed (see sub-section 3.2.4.2 dealing with such assertions).

Example:

```
TRITEM IS GROUP (STOCK#, QUANTITY);
```

indicates that the transaction item is a group (that was defined to occur 1 to 9 times) which consists of two fields, a stock number and the quantity.

3.2.3.5 Field Description

Purpose: to describe each field within each group or record of each file; to specify such information as the name of the field, its length and its attributes.

Syntax:

```
<name> IS FIELD (<field-type> (n[:m]));
```

Semantics: <name> is the name of the field, as given in the list of members in the field's parent item, which is either a group or a record;

<field type> is

(a) character (CHAR or CHARACTER),

(b) binary (BIN or BINARY),

(c) numeric character (NUMERIC), or

(d) fixed decimal (FIXED, which is IBM 370's packed decimal).

N represents the length of the field. If the field has a variable length, n is the minimum length and m is the maximum length, where 0<n<m<32768.* If the length of a field, X, is specified as varying, the user must in addition provide a "LENGTH-type assertion" which when evaluated at execution time yields the actual length of the current field (see sub-section 3.2.4.3 dealing with such assertions).

One of the primary characteristics of MODEL being a non-procedural language is that statements are independent and can appear in any order. This principle is consistent, but does have one exception which relates to the field statement. Since the same field name may and often does appear in more than one file, an assumption is made by the MODEL Processor which presumes that a field described in a FIELD Statement is associated with the file most recently described. This however, is but a slight restriction on the order-independence of statements since the user would logically want to describe each file and its component records, groups, and fields together. Furthermore, this restriction appears nowhere else. For example, assertions in the assertion section can appear in any order, the files can be described

----

* This is due to the fact that the length is stored in an IBM System-370 binary half-word.

in any order, etc.

Example:

```
NAME IS FIELD (CHAR(20));
QUANTITY IS FIELD (NUMERIC(7));
DESCRIPT IS FIELD (CHAR(1:20));
```

3.2.3.6 Interim Data Description

Purpose: to describe fields that are neither in source nor target files.

Syntax:

```
<name> IS INTERIM (<field-type> (n[:m]));
```

Semantics: In the course of specifying a problem in *MODEL*, there would be a need to express assertions involving names or variables which are described neither in source files nor in target files. When such interim variables are used, their names and attributes are described in an INTERIM statement. <name>, <field type>, n, and m are identical to the same aspects of the field description statement.

Example:

```
SALE# IS INTERIM(CHAR(5));
```

3.2.4 Assertions

Purpose: to define inter-file relationships; to define data relationships, computation rules, and decision rules; to supplement the data description with rules for computing variable length and repetition; to specify subsets of data.

Syntax:

```
<assertion name>:
    SOURCE: <name> [,<name>]* ;
    TARGET: <name> [,<name>]* ;
    "<assertion text>";
```

Semantics: The different types of assertions fall into one of several categories described in detail in sub-sections 3.2.4.1 to 3.2.4.6 below.

The <assertion name> is an arbitrary name by which the user identifies the assertion. The list of names after the keyword SOURCE are names which are sources or inputs to the relationship; i.e. are used by the assertions. The list of names (usually one) after the keyword TARGET are targets, outputs, or resultants of the computation in the assertion. The format of the <assertion text> varies for each of the different types of assertions and is described in the sub-sections below, but in general, the assertion text has the form <target-name>=<expression>.

It is true that the source and target header information for single unknown equations can be deduced from the assertion itself if a convention is maintained to place target variables always by themselves on the left of the equal(=) sign. These headers are intended for future implementations where the above rule would not be enforced. For the first proto-type version, the source and target identification of variables are intended to avoid parsing the assertion. With single unknown equations, which are handled by MODEL, the target variable can be likened to independent variables while the source variables to the dependent variables.

3.2.4.1 Inter-File Relationships: POINTER-type Assertions

Purpose: to define inter-file relationships

Syntax:

```
<assertion name>:
SOURCE: <pointing field>;
TARGET: POINTER. <record name>;
"POINTER. <record name> =<pointing field>;" ;
```

Semantics: a field in one file is said to "point to" or be a key to a record of another file when the value of the field of the first file uniquely identifies a corresponding record in the other file. For example, the stock number field (STOCK#) in the sale transaction file of the DEPSALE problem points to a corresponding record in the inventory file. Thus, the user can perceive his data base as a network of inter-related files, while the Processor utilizes the operating system's access methods for data access. <assertion name> is an arbitrary name for the assertion, <pointing field> is the name of the field which serves as the "pointer", and <record name> is the name of the record to which the field points.

The user does not declare or describe the special POINTER name as he would for data fields. The attribute of the POINTER name is assumed to be a string whose length is the same as the key field serving as a symbolic pointer. Also the POINTER name itself is not intended to be used as sources to other assertions.

Example:

```
    TRINV:
        SOURCE: TRANS.STOCK#;
        TARGET: POINTER.INVREC;
        "POINTER.INVREC=TRANS.STOCK#;" ;
```

which corresponds to the example mentioned above.

3.2.4.2 Assertions for Variable Repetition: EXIST-type Assertions

Purpose: to provide an expression that determines the number of occurrences of a variably repeating group or field.

Syntax:

<assertion name>:

```
SOURCE: <name> [,<name> ]* ;

TARGET: EXIST. <name>;

   "EXIST. <name>=<arithmetic expression>;" ;
```

<u>Semantics:</u> a group or field can occur just once, can repeat a fixed number of times, or can repeat a variable number of times. In the latter case, if an item X occurs n to m times, we write X(n:m) in the data description, where 0<=n<m. If n is zero, the group or field can exist optionally. For each group or field that may exist a variable number of times, an EXIST type assertion must be provided to indicate how many times the item exists. This assertion when evaluated at execution time yields the number of occurrences of the item. The <arithmetic expression> has the usual definition and can utilize any of the built-in functions (see Sections 3.2.4.5 and 3.2.4.7).

Special treatment and restrictions on EXIST assertions should be noted here. The user does not declare or describe the EXIST name itself as he would for data fields. The use of EXIST in an assertion implicitly gives the EXIST name a numeric attribute with a permitted value from 0 to 32767. The user has to be careful not to define EXIST.X in terms of X, which would make no sense. Furthermore, if the arithmetic expression on the right uses other fields, it must involve only fields available in the same record and to the left of the field or group whose number of occurrences is being defined. For example, the number of occurrences can be determined as the value of another field to the left of the defined group or field, or it can be determined by a delimeter to the right of the group or field being defined. Finally, other assertions may use the EXIST value as sources if the above constraints are met.

Example: if G were described to be a group with 1 to 20 repetitions, i.e. described as G(1:20), and if the actual number of repetitions were determined by the value of another field named F, then the following assertion could be used:

```
NUMG:
SOURCE: F;
TARGET: EXIST.G;
"EXIST.G=F;" ;
```

3.2.4.3 Assertions for Variable Length: LEN-type Assertions

Purpose: to provide an expression that determines the length of a variable length fields.

Syntax:

```
<assertion name>:
     SOURCE: <name> [,<name> ]* ;
     TARGET: LEN. <name>;
       "LEN. <name>=<arithmetic expression>;" ;
```

Semantics: if the length of a field is specified as varying, the length type assertion is necessary to specify how the actual length is to be computed.

The arithmetic expression has the usual definition and can utilize any of the built-in functions (see Sections 3.2.4.5 and 3.2.4.7).

Special treatment and restrictions on LEN assertions should be noted here. The user does not declare or describe the LEN name itself as he would for data fields. The use of LEN in an assertion implicitly gives the LEN name a numeric attribute with a permitted value from 0 to 32767. The user has to be careful not to define LEN.X in terms of X, which would make no sense. Furthermore, if the arithmetic expression on the right uses other fields, it must involve only fields available in the same record and to the left of the field whose length is being defined. For example, the length can be determined as the value of another field to the left of the defined field or can be determined by a delimeter to the right of the field being defined. Finally, other assertions may use the LEN value as sources if the above constraints are met.

Example: if DESCRIP were described as DESCRIP IS FIELD (CHAR(1:20)) , and if the actual length were determined by a delimiting symbol "*", then the following assertion could be used:

```
LEND:
  SOURCE: REC;
  TARGET: LEN.DESCRIP;
      "LEN.DESCRIP=DELIM(REC,'*');" ;
```

LEND is the name of the assertion; REC is the name of the record in which the delimiting symbol is being scanned; DESCRIP is the variable length field whose length is being defined by the built-in DELIM function. The parameters to the DELIM function will be explained in sub-section 3.2.4.7 dealing with functions.

3.2.4.4 Subset Selection: SUBSET-type Assertions

Purpose: to provide a way for specifying logical expressions determining which subset of a file is to be considered.

Syntax:

The name, source, and target headings as indicated in Section 3.2.4, plus the following <assertion text>:

```
IF <logical expression> THEN SUBSET. Filename = SELECTED ;
[ ELSE SUBSET. Filename = NOT_SELECTED ;]
```

Semantics: If the file name is of a source file, this statement is used to indicate which subset of the source file is to be considered as applicable to the module by means of a logical expression. This is important since different modules might want to look at different subsets of a shared file. In this case, records of the source file with name "file name", should only be considered if they satisfy the "logical expression". If the file name is of a target file, this statement is used to indicate which subset of the target file is involved in the function of the module by means of a logical expression. The logical expression can be any expression in disjunctive normal form:

<name> <relation> <value> [<operator> <name> <relation>
<value>] *

where the <name> are names of items already described; <value> is the name of another item or a constant; <relation> is any of =, >, <, <=, >=, ~=, ~<, or ~>; and <operator> is any valid combination of '&', '|', or ~.

This assertion takes the SOURCE and TARGET headings, as explained for the other assertions.

The user does not declare or describe the special SUBSET name as he does for data fields, but simply uses them in the assertions as shown in the example. The value of the subset name will be "true" or "false".

<u>Example:</u>

```
IF AGE > 20 & GRADE < 2 THEN SUBSET.STUDENT = SELECTED;
```

3.2.4.5 Assertions for Computation Rules

<u>Purpose:</u> to state data relationships and computations.

<u>Syntax:</u>

Name, source, and target headings as indicated in Section 3.2.4 plus the following <assertion text>:

```
<qname>=<arithmetic    expression>    |    <qname>=<character
expression>
<qname>::=<name> [.<name>]* [FOREACH_<name>]
```

Semantics: In the case of the arithmetic assertion, <qname> is a possibly qualified name of a numeric field (BINARY, NUMERIC, FIXED) and <arithmetic expression> has the usual definition including constants, other names, numeric functions (see Section 3.2.4.7), and arithmetic operators(+,-,*,/,**). In the case of assertions dealing with strings, <qname> is a possibly qualified name of a character field and <character expression> is some set of functions or operators on other string data (string constants or other names). Examples of operators or functions are concatenation (||) or any of the string manipulating functions such as SUBSTR (see Section 3.2.4.7 on functions).

A <name> on either the left or right side of the "=" must be qualified by prefixing it by the <name> of its parent file plus a "." whenever the field name appears in more than one file. This is to make the reference to the field unambiguous and is exemplified below.

In order to distinguish between corresponding names in a file that is both source and target of the module, the prefix "OLD." or "NEW." should be prefixed before the entire field name in order to make the distinction.

Whenever a relationship holds for each element of a repeating group or field, the word FOREACH appears in parentheses after the repeating field or group. Such uses of the FOREACH can appear on either the left or right side of the "=" sign and are exemplified in some of the sample statements below.

Example:

```
  CALCCHRG:
     SOURCE: SSLIP.SUBTOT, SSLIP.TAX;
     TARGET: SSLIP.TOTCHRG;
     "SSLIP.TOTCHRG=SSLIP.SUBTOT+SSLIP.TAX;" ;
```

CALCCHRG is the name of the assertion. The source names are SSLIP.SUBTOT and SSLIP.TAX . The target name is SSLIP.TOTCHRG. The qualifier SSLIP (sales slip) is used before the field names in this example to make the reference unambiguous with the same named fields in other files.

Another example is the following:

```
  UPDQUANT:
     SOURCE: OLD.INVEN.QOH, TRANS.QUANTITY (FOREACH_TRITEM);
     TARGET: NEW.INVEN.QOH;
     "NEW.INVEN.QOH=OLD.INVEN.QOH - TRANS.QUANTITY
     (FOREACH_TRITEM);";
```

where UPDQUANT is the name of the assertion; OLD.INVEN.QOH and TRANS.QUANTITY are the SOURCE names; NEW.INVEN.QOH is the target name. The reserved word FOREACH indicates that this relation holds for each of the components of that repeating field. In general, the FOREACH <name> is placed in parentheses after a repeating group or field, and it can appear on either the source or target sides of the assertion or on both. The reserved words OLD and NEW before a name are used to distinguish between corresponding field names in a source and target file, whenever a file is both a source and a target to the module.

3.2.4.6 Assertions for Decision Rules: CHOICE Assertions

Purpose: to state assertions that hold only under certain eventualities; i.e. to express conditional relationships. In order to facilitate this capability, MODEL's methodology lets the user express such conditional relationships in two parts shown below.

Syntax:

```
    <assertion name>:
        SOURCE: <source list>;
        TARGET: <target list>;
        "IF <logical expression> THEN CHOICE. <choice name 1> =
        SELECTED;
        [ELSE IF <logical expression> THEN CHOICE.<choice name 2>
        = selected;]*
        [ELSE CHOICE. <choice name n> = SELECTED;]*
```

```
<assertion name>:
    SOURCE: <source list>:
    TARGET: <target list>;
    "IF CHOICE.<choice name 1> THEN <assertion>;
    ELSE IF CHOICE. <choice name i > THEN <assertion>;
    ELSE <assertion>;";
```

<u>Semantics:</u> in the first part, the user has a way of assigning names (CHOICE names) to any condition. <choice name 1>, ... , <choice name n> are the names that the user assigns to each condition or CHOICE that is SELECTED. The <logical expression> is in disjunctive normal form:

```
<elementary logical expression> [<logical operator>
<elementary logical expression>]*
```

where <logical operator> is &, |, &~, or ~, and <elementary logical expression> has the form <arithmetic expression> <logical relation> <arithmetic expression>

where <logical relation> is =,<,>, <=, >=, ~<, ~>, or ~=. An example of a logical expression is

```
SALARY < 10000 & AGE > 35;
```

Secondly, after all such conditions are given names as they are selected, other assertions can reference those conditions by their "CHOICE" name. The reason for splitting up conditional relationships in this way is that other assertions can reference conditions in arbitrary order without having to repeat the logical expressions.

The user does not declare or describe the special CHOICE name as he does for data fields, but simply uses them in the assertions as shown in the examples above. The value of the choice name will be "true" or "false".

<u>Example:</u> (CHOICE selection in the DEPSALE problem deciding if the customer has exceeded his credit limit)

```
EXLIM:
    SOURCE: OLD.CUST.BALANCE, SSLIP.TOTCHRG, OLD.CUST.CREDLIM;
    TARGET: CHOICE.EXCRLIM;
    "IF OLD.CUST.BALANCE+SSLIP.TOTCHRG > OLD.CUST.CREDLIM THEN
    CHOICE.EXCRLIM = SELECTED;
```

A use of a designated CHOICE name is the following:

```
ADJ_BALC:
    SOURCE: CHOICE.SALE, OLD.CUST.BALANCE, SSLIP.TOTCHRG;
    TARGET: NEW.CUST.BALANCE;
    "IF  CHOICE.SALE=SELECTED  THEN  NEW.CUST.BALANCE  =
    OLD.CUST.BALANCE + SSLIP.TOTCHRG;";
```

It states that the new balance is equal to the old balance plus the total charge only if CHOICE.SALE has been selected. Other assertions in DEPSALE are similar.

3.2.4.7 Use of Functions

Without the use of any functions, MODEL would be limited to data processing applications where data gets transferred and relatively simple computations would be conducted. Although many data processing problems are of that nature, use of functions opens the door to much more, because functions are the language of mathematics augmenting operators.

First, we have the built-in functions of the PL/1 library such as the available mathematical functions that operate on vectors (ABS,MIN,MAX, etc.) or string manipulation functions (SUBSTR,INDEX, etc.). Any of the functions available in the PL/1 library can be used in an assertion. Their use is documented in the PL/1 reference manual [PL1 75].

Secondly, a capability to use functions allows a systems programmer (in this context, a procedural programmer who would maintain the MODEL processor) to add power to the System by adding functions to a library. Since MODEL assertions are limited to arithmetic functions of the form Y = F(X1,X2,...,Xn), adding functions would enhance MODEL's "knowledge". By definition, MODEL in being a non-procedural language, will always be incomplete; i.e. there will always be a computation which cannot be expressed non-procedurally.

This, however, does not detract from the value of the language such as MODEL for two reasons. First, the class of problems which actually can be expressed is so large that it covers most data processing applications. Second, by adding appropriate functions to a library, the system's "knowledge" can be increased. It is anticipated that with a supply of several dozen of the most common functions used in data processing, almost any data processing problem could be expressed non-procedurally in MODEL.

Examples of useful functions in many data processing applications that could be added to the library are summation, minimum, maximum, average, and median.

Some functions have already been written and put into a library (called FCNS) here. They are named DELIM, REPLACE, SERIAL#, SUM, and TODAY, and are exemplified in the DEPSALE problem. The use of each of these functions is summarized below. Their actual source code is provided in Appendix B along with the Processor modules. The Processor incorporates the text of those functions actually used by a specification by including them in the generated program. An aleternative way of adding functions would be to place them in object form, appended to the PL/1 library by using, for example, IBM's catalogued procedure PL1CL [PL1 75]. Those new functions already provided are summarized here.

Functions: DELIM

Purpose: to scan for a specified delimiter.

Parameters: (string-name, delimiter)

string name: the name of the string being scanned; i.e. the name of the input record)

delimiter: the character(s) that delimit the variable-length field

Result: the number of characters from the beginning of the current field up to but not including the specified delimiter is returned.

Function: REPLACE

Purpose: to stack a vector of replacements or alternatives for successive use.

Parameters: (vector-name, number)

vector name: name of the list of replacements

number: the number of occurrences in the vector or the EXIST name that currently has the number of occurrences.

Result: See Section 3.2.4.8 for a detailed discussion of the REPLACE function.

Function: SERIAL#

Purpose: to produce serial numbers

Parameters: (increment, counter#)

increment: the amount by which each number in the series is to be incremented

counter: a number to distinguish between different series of serial numbers

Result: the next number in the series identified by "counter#" is returned, differing from the previous number by "increment" amount

Function: TODAY

Purpose: to provide today's date

Parameters: none

Result: the current date is returned in the form "mmm dd, yy", where mmm is the three letter abbreviation of the month, dd is the day, and yy is the year.

Function: SUM

Purpose: to add numbers of a vector (repeating fields); not to be confused with another summation function (SUMMAT) alluded to elsewhere with the purpose of adding values of a field across records.

Parameters: (vector name, index, lower bound, upper bound)

vector name: name of repeating field whose sum is desired

index: the FOREACH name corresponding to the repeating field

lower bound: element of repeating field with which total starts

upper bound: element of repeating field with which total ends

Result: the total of the elements of the vector is returned.

Some functions do not complete their computation upon each invocation, but only when an associated condition is met. That is, some functions can have conditions associated with them which determine their completion. A summation function, for example, could complete adding up all the desired values of a field in different records only when it reaches the last item to be included in the total. The MODEL Processor maintains a table of such functions for its own use described in Chapter 4.

Since the Processor does not scan the text of the assertion for reasons already explained, and since it relies on the heading of each assertion in the current version, the user is required to add the *following to the heading* (following the SOURCE and TARGET heading) whenever a function is used in the assertion:

```
        FUNCTION: function name;
```

This would appear immediately after the SOURCE and TARGET heading such as in the example below:

```
        TOTX:
            SOURCE: X,Y;
            TARGET:Z;
            FUNCTION: SUM;
            "Z=SUM(X,Y,...);";
```

3.2.4.8 Allowable Cycles: the REPLACE Function

It is clear that the set of assertions provided by the user must be complete, consistent, unambiguous and otherwise well-defined. This will be dealt with in more detail in Chapter 4. Specifically, the assertions cannot be circular. For example, A being the source of B, B being the source of C, and C being the source of A would cause obvious problems. Although such cycles are generally illegal (and are detected by the Processor as will be explained in Chapter 4), and although the philosophy of the MODEL Language avoids any expression of control structures, there is one exception. There is a type of circumstance where one would want a set of assertions reasserted, but based on different values. For example, in the DEPSALE problem we would want to have a rule that if a desired stock item is out of stock, the transaction can be completed by providing a substitute for the desired item and proceeding with all the assertions that define the transaction, but with the replaced substitute item. This was done in the sample problem by replacing the value of the substitute item as a pointer to the inventory file. In MODEL this assertion is written as follows in the DEPSALE problem:

```
TRYREPL:
     SOURCE: OLD.INVEN.SUBST#, CHOICE.SUBSTIT, EXIST.SUBST#;
     TARGET: POINTER.OLD.INVREC;
     FUNCTION: REPLACE;
     "IF   CHOICE.SUBST=SELECTED   THEN   POINTER.OLD.INVREC  =
     REPLACE(OLD.INVEN.SUBST#, EXIST.SUBST#);";
```

where REPLACE is a function meaning that all assertions requiring POINTER.OLD.INVREC and its consequences should be re-evaluated with the substituted number.

The user would, of course, have to ensure that the replacement or substitution will either not repeat endlessly and that eventually a substitute item will take a choice which does not cause another replacement (in the example if a substituted item would be sufficiently in stock) or he would specify a special condition (called EMPTY) asserting a relationship in the eventuality that none of the replaced items would select a different choice (in the example, if none of the substitute items were in stock). An example of such an "empty list" condition is the following:

```
REORDER:
    SOURCE: CHOICE.EMPTY;
    TARGET: SALECODE;
    "IF CHOICE.EMPTY THEN SALECODE='RO';";
```

This signifies that the message should be indicated as "reorder" (RO) if the list of substitute items is empty; i.e. they all failed to complete a sale successfully without selecting a choice other than replacement. While such a specification for replacement is somewhat contrary to the non-procedural philosophy of all the other MODEL statements, it does provide a useful tool in some data processing applications.

The actual code for the REPLACE function is provided in the appendix.

The next section presents a formal specification of the MODEL language. Chapter 4 that follows describes the design, methodology, and algorithms that enable processing of specifications written in MODEL.

3.3 The MODEL Language Described in EBNF

This section presents a formal complete description of the syntax of MODEL in an Extended Backus Normal Form (EBNF) specification language [RAM73]. In the grammar of MODEL that follows, angle-bracketed names < > represent syntactic names and non-bracketed names represent terminal symbols, as in conventional BNF. Units enclosed in square brackets [ ] indicate that they are optional, and an asterisk following square brackets [ ]* indicates repetition zero or more times. Also, level numbers are indicated for readability to show the depth within the tree structure. Only the syntactic composition of each statement is shown here. The semantic constraints associated with the overall specification and the actions performeu by the Processor upon recognition of each syntactic unit are subjects dealt with in Chapter 4.

```
1    1 <MODEL-SPECIFICATION>::=[<MODEL-BODY-STMTS>;]*
2      2 <MODEL-BODY-STMTS>::=<MODULE-NAME-STMT>
3          | <SOURCE-FILES-STMT>
4          |<TARGET-FILES-STMT>
5          |<DATA-DESC-STMT>
6          |<ASSERTION>
7          |END
8        3 <MODULE-NAME-STMT>::= MODULE: <NAME>
9        3 <SOURCE-FILES-STMT>::=SOURCE FILES:<NAMELIST>
10       3 <TARGET-FILES-STMT>::=TARGET FILES:<NAMELIST>
11         4 <NAMELIST>::=<NAME>[,<NAME>]*
12       3 <DATA-DESC-STMT>::=<NAME>IS<DATA-DESC>
13         4 <DATA-DESC>::=<FILE-STMT>
14             |<RECORD-STMT>
15             |<GROUP-STMT>
16             |<FIELD-STMT>
17             |<REPORT-STMT>
18             |<REPORT-ENTRY-STMT>
19             |<INTERIM-STMT>
20             |<STORAGE-STMT>
21           5 <FILE-STMT>::=FILE(RECORD IS <NAME>[,STORAGE IS <NAME>] [,
             <KEY> IS <NAME>])
22             6 <KEY>::=KEY|SEQUENCE
23           5 <RECORD-STMT>::=RECORD(<ITEM-LIST>)
24           5 <GROUP-STMT>::=GROUP(<ITEM-LIST>)
25             6 <ITEM-LIST>::=<ITEM> [,<ITEM>]*
```

```
26          7 <ITEM>::=<NAME>[(<INTEGER>[:<INTEGER>])]
27        5 <FIELD-STMT>::=FIELD
           (<TYPE>(<INTEGER>[:<INTEGER>]))
28         6 <TYPE>::=CHAR|CHARACTER|NUMERIC|BIN|BINARY| FIXED
29        5      <REPORT-STMT>::=REPORT(REPORT_ENTRY      IS       <NAME>
          [,<KEY>IS<NAME>])
30        5 <REPORT-ENTRY-STMT>::=REPORT_ENTRY(<ITEM-LIST>)
31        5 <INTERIM-STMT>::=INTERIM
           (<TYPE>(<INTEGER>[:<INTEGER>]))
32        5 <STORAGE-STMT>::=CARD
33            | PUNCH
34            | PRINTER[(LINE_LENGTH=<INTEGER>)]
35            | TAPE(<RECORD-FORMAT>,VOL_NAME=<NAME>
36              [,INT_NAME=<NAME>]
37              [,NO_TRKS=<NO-TRKS>]
38              [,PARITY=<PARITY>]
39              [,DENSITY=<DENSITY>]
40              [,TAPE_LABEL=<TAPE-LABEL>.  ]
41              [,START_FILE=<INTEGER>])
42            |DISK([ORG=<ORG-TYPE>]
43              <RECORD-FORMAT>
44              [,VOL_NAME=<NAME>]
45              [,INT_NAME=<NAME>]
46              [,UNIT=<TYPE-DISK>]
47              [,SPACE=<UNITS>,<INTEGER>[,<INTEGER>]])
48            |TERMINAL(<RECORD-FORMAT>
          ,TERMNAME=<NAME>
          [,UNIT=<NAME>])
     6 <NO-TRKS>::=7 | 9
     6 <PARITY>::=ODD | EVEN
     6 <DENSITY>::=200 | 556 | 800 | 1600
     6 <TAPE-LABEL>::=IBM_STD,INT_NAME=<NAME>
         |ANSI_STD,INT_NAME=<NAME>
         | NONE
         | BYPASS
     6   <ORG-TYPE>::=SEQUENTIAL    |    ISAM    |    INDEXED    |
         INDEXED_SEQUENTIAL
     6 <TYPE-DISK>::=2314 | 2311 | 3330 | 2305
     6 <UNITS>::=TRACKS|CYL|<INTEGER>
     6 <RECORD-FORMAT>::=FIXED , BLOCKSIZE=<INTEGER>
           [,RECORDSIZE=<INTEGER>]
         |VARIABLE , MAX_BLOCKSIZE=<INTEGER>
           [, MAX_RECORDSIZE=<INTEGER>]
         |VARIABLE_SPANNED , MAX_BLOCKSIZE=<INTEGER>
           [,MAX_RECORDSIZE=<INTEGER>]
         |UNDEFINED , MAX_BLOCKSIZE=<INTEGER>
      7 <INTEGER>::=<DIGIT>[<DIGIT>]*
      7 <DIGIT>::=0|1|...|8|9
  3 <ASSERTION>::=<NAME>:
        SOURCE:<ONAME>[,<ONAME>]*;
        TARGET:<QNAME>[,<QNAME>]*;
        [FUNCTION:<NAME>;]
```

```
            "<TEXT>"

      4 <QNAME>::=<NAME>[.<NAME>]*

             [(FOREACH_<NAME>)]

        5 <NAME>::=<LETTER>[<CHAR>]*

         6 <CHAR>::=<LETTER>|<DIGIT>| @ | # | _

           <LETTER>::=A|B|...|X|Y|Z
```

CHAPTER 4

The MODEL Processor

4.1 Overview

This chapter describes the algorithms and mechanisms of the MODEL Processor, which is a software system performing the program writing function. As was explained in Section 3.1.1, the MODEL Processor (hereafter called the Processor) has been designed in order to automate the program module design, coding, and debugging phases of software development based on module specifications in the non-procedural MODEL language. It presupposes that the functions of the desired application have been described to the extent that the file, inter-file, and intra-record structures have been described and that the system has been partitioned into functional modules. As shown in Figure 4.1, a module is then formally described and specified in the MODEL language, whose statements are then submitted to the Processor. Each MODEL statement composed by the user is referred to as a description, while the set of MODEL statements describing a functional module is referred to as a specification. The Processor, in turn, performs the analysis (including checking for the completeness and consistency of the descriptions and the entire specification), module design (including generating a flowchart-like sequence of events for the module), and code generation functions, thus replacing the tasks of the application programmer/coder. The Processor's capability to process a non-procedural specification language is built on an application of graph theory to the analysis of such specifications and to the program generation task.

```
                    +---------------------+
MODEL Statements    | MODEL Processor     |--> PL/1 Program
                    |                     |
for each----------->|     Spec Anal       |
Functional Module   |     Program Design  |
                    |     Coding          |--> User Reports
                    +---------------------+
```

Figure 4.1

Overview of MODEL Processor

Another important function of the Processor is to interact with the specifier to indicate necessary supplements or changes to the submitted statements.

The Processor produces a complete PL/1 program ready for compilation and execution by the object operating system as well as various reports concerning the specification and the generated program. The Processor output reports include a listing of the specification, a cross-reference report, a flowchart-like report on the generated program, and a listing and summary of the generated program, all to be described fully later. These are regenerated whenever a specification is submitted.

Processing of the module specification written in MODEL by the Processor consists of five phases shown in the system flowchart of Figure 4.2, which is the first refinement of Figure 4.1. Some of these phases represent adaptions of known but state-of-the-art technology, while some of the latter phases involve more novel innovations in analysis of the specification and in the design and code-generation for the application program.

Each of the five phases depicted in Figure 4.2 is discussed below.

Phase (1) Syntax Analysis of the MODEL Module Specification

In this phase, the provided MODEL Specification is analyzed to find syntactic and some semantic errors. This phase of the Processor

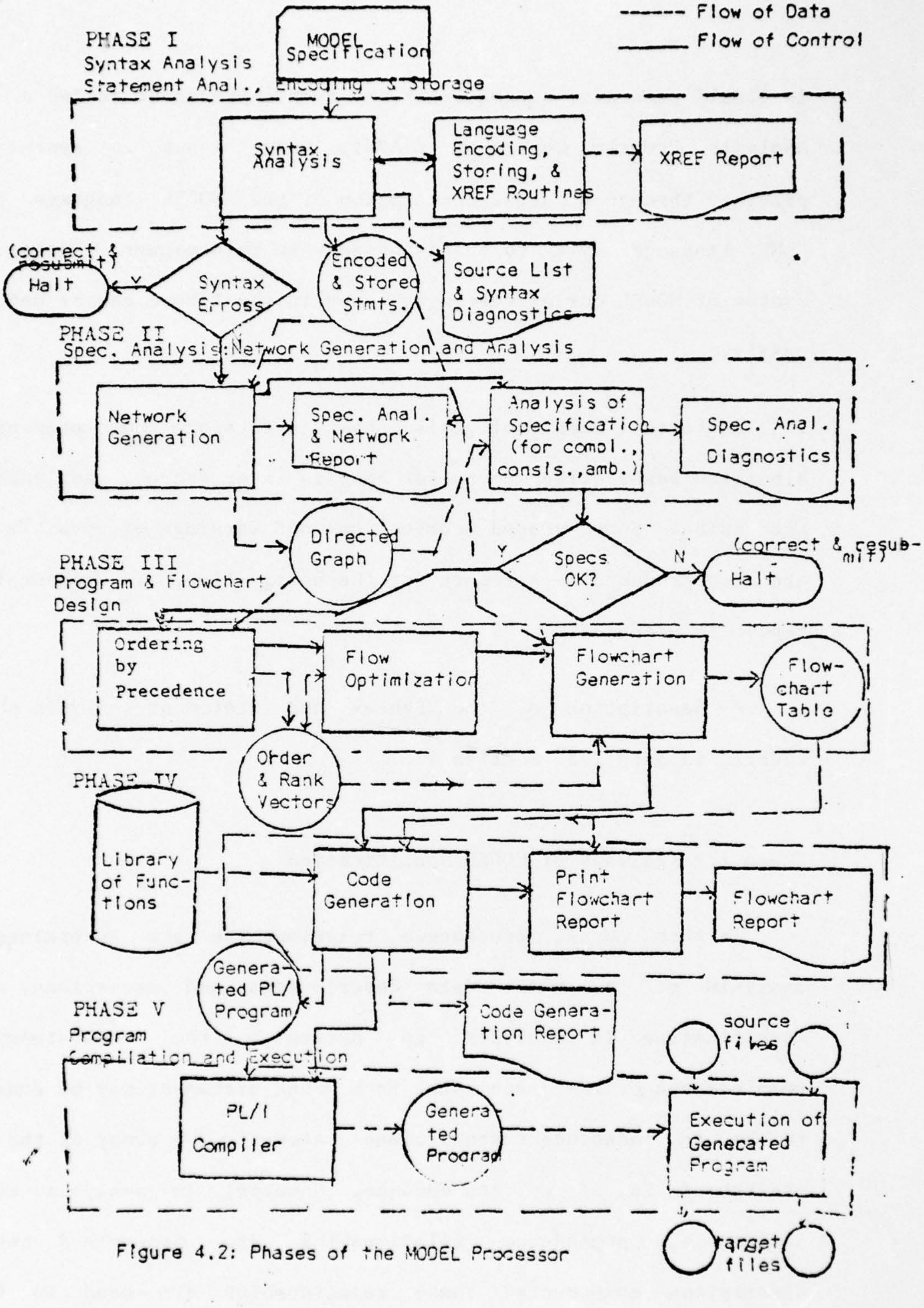


Figure 4.2: Phases of the MODEL Processor

is itself generated automatically by a meta-processor called a Syntax Analysis Program Generator (SAPG), whose input is syntax rules provided through a formal description of the MODEL language in the EBNF language (yet to be discussed). In this manner, changes to the syntax of MODEL during development and in the future can be made more easily.

A further task of this phase is to store the statements in a simulated associative memory for ease in later search, analysis, and processing. Some needed corrections and warnings of possible errors are also produced in a report for the user. Also, a cross-reference report is produced.

A description of the Syntax and Statement Analysis phase is covered in detail in Section 4.2.

Phase (2) Analysis of MODEL Specification

In this phase, precedence relationships are determined from analysis of the MODEL data descriptions and assertions, and the specification is analyzed to determine the consistency and completeness of the statements. Each MODEL statement may be considered to be an independent stand-alone statement. The order of the user's statements is of no consequence. However, in analysis of the statements, precedence relationships are determined based on description components. These relationships are used to form a precedence graph on which the completeness, consistency, ambiguity, and feasibility of the specification can be checked. Reports are produced for the user indicating the data, assertions, or decisions that have been inadequately described, assumptions that have been made by the Processor, or contradictions that have been found, and reports are provided to cross-check the descriptions.

Explanation of this process is covered in Section 4.3.

Phase (3) Automatic Program Design and Generation of Sequence and Control Logic.

This phase of the Processor determines the sequence of execution of all events implied by the specification, using precedence and graph theory techniques, and thereby determines the sequence and control logic of the desired module. Design of the object program proceeds with scope and iteration analysis and flow optimization. The result of this phase is a set of data structures representing the desired sequence of processes and flow of events, sequenced, ranked, and optimized in their order of execution. Thus, the output of this phase is a table that is similar to a program flowchart of the desired module, and is subsequently used to produce a flowchart-like report. This phase is presented in detail in Section 4.4.

Phase (4) Code Generation

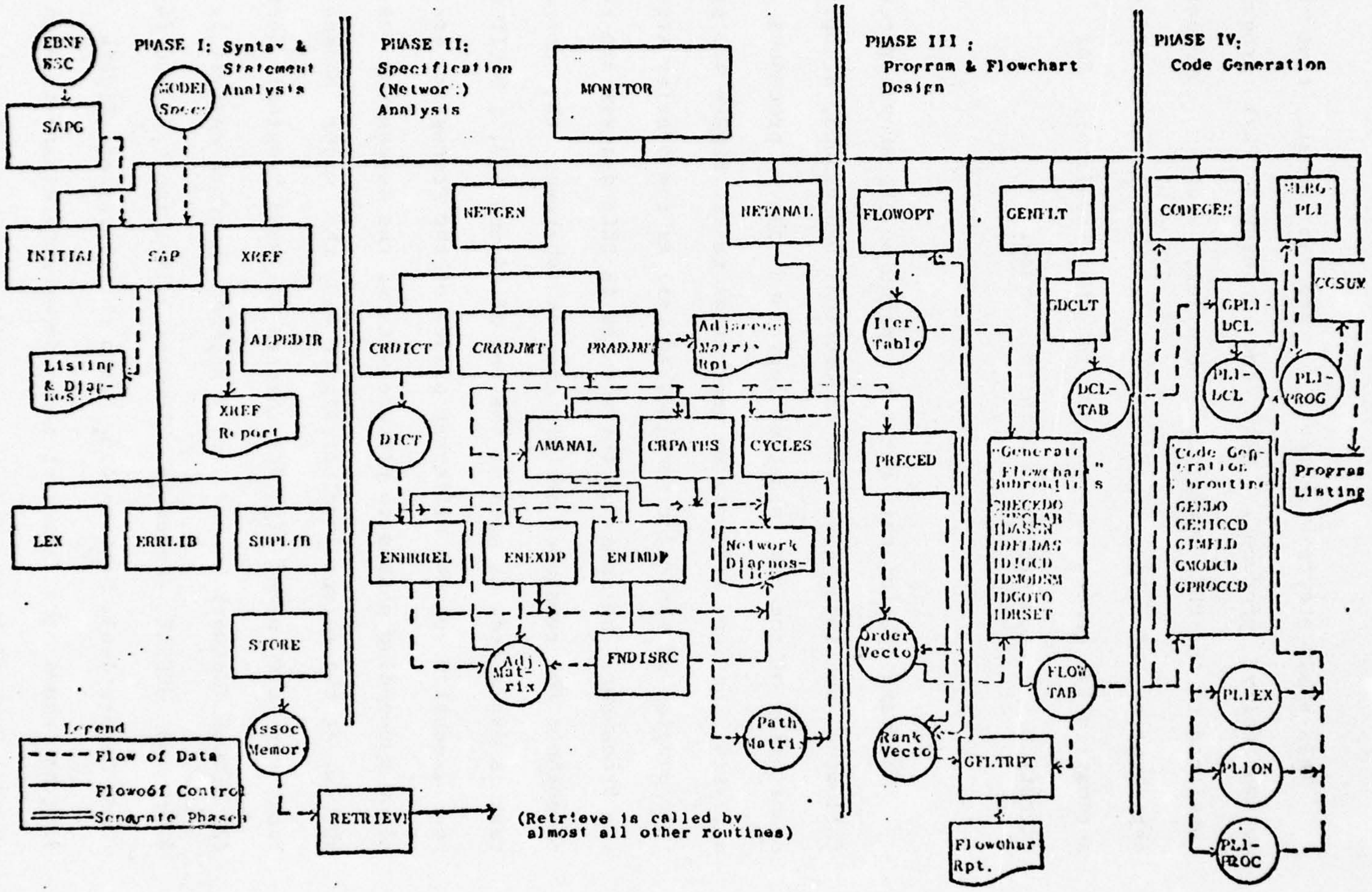
At this point in the process it is necessary to generate, tailor, and insert the code into the entries of the flowchart to produce the program. Code is produced in two steps for purposes of modularity and independence of the target language. The first step produces a language-independent version of the flowchart-entries, as noted above, while this second step produces code in the PL/1 programming language. In particular, read and write input/output commands are generated whenever the flowchart indicates the need for records. Calls to procedures embodying the assertions are generated in the appropriate places in the flowchart. Wherever program iterations and other control structures are necessary, program code for them is generated. Declarations for object program data structures and variables are generated. The product of this phase is a complete program in a high level language, PL/1, ready for compilation and execution. A listing of the generated program as well as the flowchart-like report is produced. This documentation is not expected to be of significance to the casual user, but would be available for a computer programmer in the event that it may be needed for deeper understanding or debugging. Just as the present high level PL/1 programmer does not refer to the assembly language listing which may be a by-product of the conventional compiler, so the future analyst using a MODEL type system would not refer to intermediate level programs that would be by-products. A further product of this phase is the JCL statements for the operating system. The code generation phase is covered in Section 4.5.

Phase (5) Program Compilation and Execution

This phase starts with the PL/l program module that has been automatically produced. With the generated PL/l program being submitted to the PL/l optimizing compiler, program compilation and optimization of code on the machine-language level is effected. The automatically generated program is then available for use in execution. Section 4.6 deals with this phase.

The remainder of this chapter expands on the above processes. The following sections give the theoretical background, algorithms, and description of the system methodology and above processes for the analysis, design, and program generation tasks. Figure 4.3 provides a tree diagram of the major modules, as well as the overlay structure of the Processor. The names of the modules in this diagram are referenced throughout the remainder of this chapter wherever the corresponding task is explained. As seen at the top of Figure 4.3, a MONITOR governs the execution of the different phases of the Processor, and does not allow succeeding phases to proceed without the success of the previous phases. At the second level of Figure 4.3, the major phases of the Processor are named (1) SAP (Syntax Analysis Program), Section 4.2; (2) NETGEN (Network Generation) & NETANAL (Network Analysis), Section 4.3; (3) GENFLT (Generate Flowchart), Section 4.4; and (4) CODEGEN (Code Generation), Section 4.5. Below this level of Figure 4.3, the diagram shows the names of the modules subordinate to each of these phases. Each of these subroutines is discussed at length throughout

Figure 4.3 Major Modules of the MODEL Processor



Major Modules	Section
ALPHDIR	4.2.6
AMANAL	4.3.3.8
CHECKDO	4.4.3.7
CHECLAB	4.4.3.8
CGSUM	4.5.2
CODEGEN	4.5, 4.5.1
CRADJMT	4.3.3.2
CRDICT	4.3.3.1
CRPATHS	4.3.3.9
CYCLES	4.3.3.9
ERRLIB	4.2.2.3
ENEXDP	4.3.3.4, 4.3.3.5
ENHRREL	4.3.3.3
ENIMDP	4.3.3.6
ENPTREL	4.3.3.7
FLOWOPT	4.4.2
FNDISRC	4.3.3.6
GDCLT	4.4.3.11
GENDO	4.5.1.6
GENFLT	4.4.3, 4.4.3.1
GENIOCD	4.5.1.2, 4.5.1.3
GFLTRPT	4.4.3.9
GIMFLD	4.5.1.8
GMODCD	4.5.1.1
GPLIDCL	4.5.1.11
GPROCCD	4.5.1.4, 4.5.1.5
IDASSN	4.4.3.5
IDFLDAS	4.4.3.6
IDIOCD	4.4.3.3, 4.4.3.4
IDMODNM	4.4.3.2
IDGOTO	4.4.3.9
IDRSET	4.4.3.9
LEX	4.2.2.1
MERGPL1	4.5.3
NETANAL	4.4
NETGEN	4.3.2
PRADJMT	4.3.2.4
PRECED	4.4.1
RETREVE	4.2.4.5
SAP	4.2.1
SAPG	4.2.1
STORE	4.2.4.4
SUPLIB	4.2.2.2, 4.2.2.4, 4.2.2.5
XREF	4.2.6

Figure 4.3a

Index of Major Modules

this chapter. Figure 4.3a provides an alphabetic index of the names of the major modules and the sections in which they are discussed.

In order to exemplify the nature of the Processor phases throughout the analysis, design, and program generation phases, a sample case problem is described in Section 4.3 and specified in MODEL. The processing of that sample problem (a subset of the Department Store Sale problem of Chapter 3) is followed throughout the various phases for tutorial purposes.

One final note is that the reader might wish to skip some of the following sections of the Processor description which can be done depending on the level of familiarity or detail of understanding desired. For example, the reader might wish to skip the sections on lexical and syntax analysis if he is already familiar with the techniques used here, and go on to the heart of the Processor, phases 2 through 4.

4.2 Syntax and Statement Analysis

The first phase of processing of MODEL statements is syntax and other analysis local to statements, described in the following subsections. While syntax analysis technology is well-known, advanced state-of-the-art techniques not only have been used here, but also proved to be invaluable. Specifically, the capability to generate the parser automatically has enabled rapid development changes. In addition to checking the MODEL statements for syntactic and some semantic errors, this phase also stores the specification in an internal associative form for further processing.

4.2.1 EBNF, SAPG , and the Syntax Analysis Program

4.2.1.1 Specification of MODEL using EBNF and the SAPG.

The Syntax Analysis Program (SAP) for the MODEL statements is mostly generated automatically by a Syntax Analysis Program Generator (SAPG). The SAPG used is the one developed by the University of Pennsylvania DDL Project (of which the author was a member). Documentation of its design can be found in [RAM73] and of its implementation in [FRE72]. As shown in Figure 4.4, the SAPG produces the Syntax Analysis Program (SAP) for analyzing MODEL statements, by inputting a specification of the MODEL language in the meta-language "Extended Backus Normal Form with Subroutine Calls" (EBNF/WSC) .

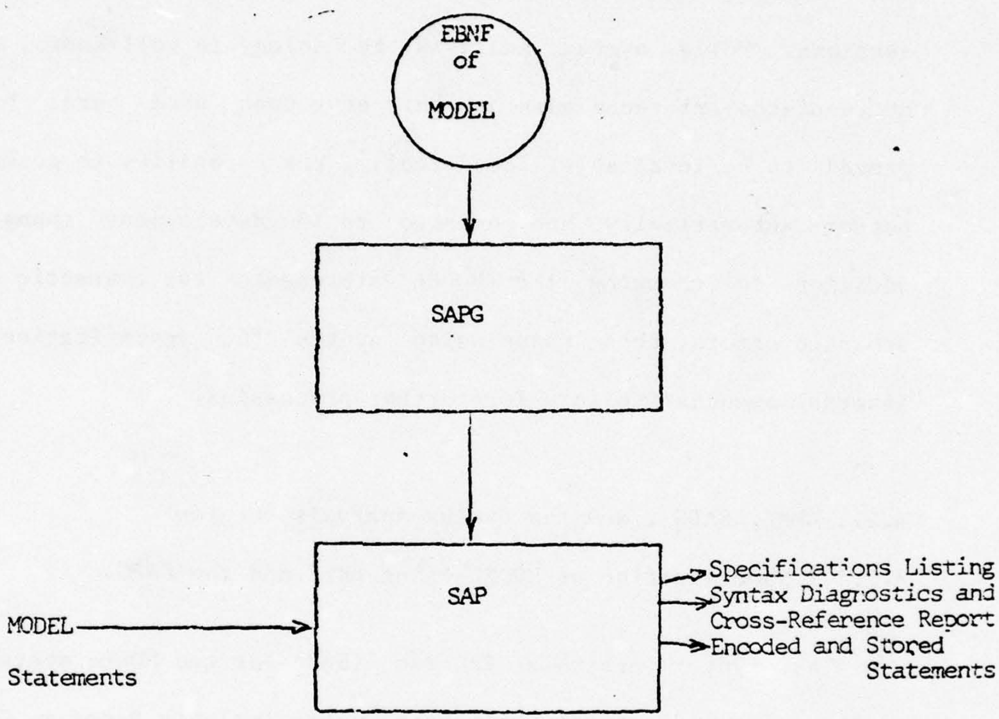

EBNF of MODEL
SAPG
SAP
MODEL Statements
Specifications Listing
Syntax Diagnostics and Cross-Reference Report
Encoded and Stored Statements

Figure 4.4

Block Diagram of SAPG and SAP

The specification of the MODEL language uses the EBNF/WSC meta-language which is submitted to the SAPG. The EBNF/WSC meta-language, which is used here to describe MODEL, was developed with the SAPG as described in [RAM73, FRE72]. The nature of the EBNF/WSC meta-language is reviewed here to make its use for MODEL complete.

The EBNF/WSC includes the traditional concepts of BNF. BNF uses sequences of characters enclosed in angle-brackets < > called <u>non-terminals</u> to give names to grammatical units and for which substitutions may be made. It also uses sequences of characters not enclosed in brackets which are in the object language (in this case MODEL). BNF consists of a series of <u>production rules</u> or <u>substitution rules</u> of the form "A::=B". "A" is a single non-terminal symbol and "B" is one or more alternative sequences of terminal or non-terminal symbols that can be substituted for A. The alternatives are separated by the meta-symbol "|". To facilitate language description, BNF was extended to EBNF with two more well-known meta-symbols: [ ] representing optionality and [ ]* representing repetition zero or more times (the quotation marks " are used in this implementation instead of square brackets because of their greater availability on standard keypunches).

A description of the MODEL language using EBNF (without subroutine calls) was provided in Section 3.3 of Chapter 3. As seen there, EBNF is sufficient to describe the syntax of the object language MODEL.

Actually, the specification of MODEL that is input to the SAPG consists not only of the syntax specification of MODEL, but also of subroutine names embedded within the EBNF; therefore the name "ENBF with Subroutine Calls" (EBNF/WSC). The EBNF/WSC provides a capability to branch to these subroutines upon successful recognition of a syntactic unit. Thus, they can complete the SAP to enable it to check some of the statement semantics, to encode, to produce error messages, and to store the MODEL statements for later retrieval. The invocations of these subroutines are generated automatically by the SAPG, while the supporting subroutines themselves are written manually. The specification of the MODEL language in the EBNF/WSC meta-language, which is submitted to the SAPG appears in Figure 4.5. Unlike the human-oriented EBNF of Chapter 3, this one has two machine-oriented modifications:

(1) the names of the invoked subroutines are embedded in the EBNF (enclosed in slashes);

(2) the EBNF itself has been restructured to conform to restrictions explained in the next section that are imposed by the SAPG Processor.*

---

* The grammar must be of type "LR-1", which means that the grammar is to be written in such a way that backtracking is limited to one token at each level in the tree. Writing the grammar in "LP-1" form is made possible for MODEL by inserting many keywords.

Figure 4.5
EBNF/WSC for MODEL

```
EBNF EBNF
WSC  Ref.
Stmt No.

1     1     <MODEL_SPECIFICATION>::= "<MODEL_BODY_STMTS>  /CLRERRF/"*
            /STMT_FL/ <MODEL_SPECIFICATION>

2     2    <MODEL_BODY_STMTS>::= /UNRECS/
      3.1     |MODULE <MODULE_NAME_STMT>
      4.1     |SOURCE <SOURCE_FILES_STMT>
      5.1     |TARGET <TARGET_FILES_STMT>
      6.1     |<HDCS>
              |<NAME> <DDL_OR_ASSER_STMT>
     30.1     |END /ENDINP/

3     3.2     <MODULE_NAME_STMT>::=  /MODUL1/: /MODUL2/ <NAME> /STMOD/
            <ENDCHAR>

4     4.2  <SOURCE_FILES_STMT>::= "<FILE_KEYWORD>" /SRCFL1/ /INITSFL/
            : <SOURCE_FILELIST> /STSRC/ <ENDCHAR>

5     4.3  <FILE_KEYWORD>::=FILES|FILE

6     4.4  <SOURCE_FILELIST>::= /SRCFL2/ <NAME> /SVSRC/  ",  /SRCFL2/
            <NAME> /SVSRC/" *

7     5.2  <TARGET_FILES_STMT>::= "<FILE_KEYWORD>" /TARFL1/ /INITTFL/
            : <TARGET_FILELIST> /STTAR/ <ENDCHAR>

8     5.3  <TARGET_FILELIST>::= /TARFL2/ <NAME> /SVTAR/
            ". /TARFL2/ <NAME> /SVTAR/ " *
```

```
9     6.2,30.2        <DDL_OR_ASSER_STMT>::=       /SVDDNM/       /BADDDS2/

              <DDL_OR_ASSER_BODY>

10    6.3,30.3            <DDL_OR_ASSER_BODY>::=            <DATA_DESC_STMT>

              |<ASSERTION_DESC>

11    6.4          <DATA_DESC_STMT>::=  IS   /BADDDS/  <DATA_DESCRIPTION>

              <ENDCHAR>

12    7          <DATA_DESCRIPTION>::=

              <FILE_DESC_STMT>

                |<STORAGE_DESC_STMT>

                |<RECORD_STMT>

                |<GROUP_STMT>

                |<FIELD_STMT>

                |<REPORT_STMT>

                |<REPORT_ENTRY_STMT>

                |<INT_DESC>


13    8.1         <FILE_DESC_STMT>::= FILE /SVFILE/ /FILERR1/

              ( <FILE_SPECIFICATION> ) /STFILE/

14    8.2         <FILE_SPECIFICATION>::= /FILERR2/  <FILE_DESCRIPTION>  |

              <LIB_CALL>

15    8.3           <FILE_DESCRIPTION>::=  RECORD  "IS"  /FILERR3/  <NAME>

              /SVRCNM/ ","<FILE_STOR>

16    8.4         <FILE_STOR>::=

                "CHAR_CODE "IS" /FILERR4/ <CODE> /SVCC/" ","

                "STORAGE "IS" /FILERR5/ <NAME> /SVSTNM/ " ","

                "<KEY> "IS" /FILERR6/ <NAME> /SVKEY/ "
```

```
17     8.5      <LIB_CALL>::=LIBRARY  /LIBERR/  (  <NAME>  /SVLBNM/   )
               /GETLIB/
18     8.6     <CODE>::=EBCDIC|BCD|ASCII
19     9       <KEY>::=KEY|SEQUENCE
20     10      <RECORD_STMT>::= RECORD /MEMINIT/ /RECERR/ (<ITEM_LIST> )
               /STREC/
21     11       <GROUP_STMT>::=  GROUP /MEMINIT/ /GRPERR/ (<ITEM_LIST>)
               /STGRP/
22     12      <ITEM_LIST>::=<ITEM> ", <ITEM>" *
23     13.1    <ITEM>::= /ITEM01/ <NAME> /SVMEM/
               "(/MINERR/ <MINOCC> /SVMNOC/ <OCC_END>"
24     13.2    <OCC_END>::=) |:/MAXERR/ <MAXOCC> /SVMXOC/ /CKMNMX/)
25     13.3   <MINOCC>::= <INTEGER>
26     13.4   <MAXOCC>::=<INTEGER>
27     14.1    <FIELD_STMT>::=FIELD /SVFLD/ <FIELD_ATTR> /STFLD/
28     14.2     <FIELD_ATTR>  ::=  /FLDERR1/   (   <TYPE>   /SVFDTP/   (
               <MIN_LENGTH>
               /SVMNFLN/  ":  /FLDERR2/  <MAX_LENGTH> /SVMXFLN/ " ) ","
               "<LINE_SPEC>" "," "<COL_SPEC>" )
29     14.3   <LINE_SPEC>::= LINE /LCERR/ (<INTEGER> /SVLINE/)
30     14.4   <COL_SPEC>::= COL /LCERR/ (<INTEGER> /SVCOL/)
31     14.4   <MIN_LENGTH>::= <INTEGER>
32     14.5   <MAX_LENGTH>::= <INTEGER>
33     15             <TYPE>::=          CHAR|BIN|CHARACTER|BINARY|FIXED
               |DECIMAL|NUMERIC " TABULATED /SVTAB/ " /STGRP/
```

```
34     16      <REPORT_STMT>::= REPORT  /SVRPT/ /RPTERR/ (REPORT_ENTRY
              "IS" <NAME> /SVRENM/ "," <FILE_STOR> ) /STRPT/

35     17     <REPORT_ENTRY_STMT>::=REPORT_ENTRY /MEMINIT/  /RPTNER/  (
              <ITEM_LIST> ) /STRPTN/

36     18     <INT_DESC>::= INTERIM /SVINMM/ <FIELD_ATTR> /STINT/

37     19.1     <STORAGE_DESC_STMT>::=   <CARD_STMT>  |  <TAPE_STMT>  |
              <DISK_STMT>   |   <PRINTER_STMT>   |   <PUNCH_STMT>    |
              <TERMINAL_STMT>

38     19.2  <CARD_STMT>::=CARD /STCARD/

39     19.3  <PRINTER_STMT>::= PRINTEP /STPRNT/

40     19.4  <PUNCH_STMT>::= PUNCH /STPNCH/

41     19.5     <TERMINAL_STMT>::=   TERMINAL    /SVTERM/    /TERMERR/  (
              <TERM_DESC_BLOCK>) /STTERM/

42     19.6   <TERM_DESC_BLOCK>::= <RECORD_FORMAT> "," <TERM_NAME_SPEC>
              "," "UNIT "=" /DSKER4/ <ANY> /SVTMUN/ "

43     19.7   <TERM_NAME_SPEC>::= /VOLERR/ TERMNAME "=" <ANY> /SVTRMNM/

44     19.8  <ANY>::= <NAME> | <INTEGER>

45     19.9   <TAPE_STMT>::= TAPE /SVTAPE/ /TAPERR/  (  <RECORD_FORMAT>
              "," <VOL_NAME_SPEC> ","
              "<INT_LABEL_SPEC> ","
              "<NO_TRKS_SPEC>" ","
              "<PARITY_SPEC>" ","
              "<DENSITY_SPEC> ","
              "<TAPE_LABEL_SPEC>" ","
              "<START_FILE_SPEC>" "," ) /STTAPE/
```

```
46    19.10   <VOL_NAME_SPEC>::= /VOLERR/ VOL_NAME "=" <NAME> /SVVOL/

47    19.11   <INT_LABEL_SPEC>::=  INT_NAME  ","  /PARERR/  <NAME>
              /SVINTNM/

48    19.12   <NO_TRKS_SPEC>::= NO_TRKS "=" /PARERR/ <NO_TRKS> /SVTRK/

49    19.13   <PARITY_SPEC>::= PARITY "=" /PARERR/ <PARITY> /SVPAR/

50    19.14   <DENSITY_SPEC>::= DENSITY "=" /PARERR/ <DENSITY> /SVDEN/

51    19.15   <TAPE_LABEL_SPEC>::=   TAPE_LABEL   "="   /PARERR/
              <TAPE_LABEL>

52    19.16   <START_FILE_SPEC>::=  START_FILE /PARERR/ "=" <INTEGER>
              /SVSTFL/

53    19.17   <DISK_STMT>::= DISK /SVDSK/ /DSKER1/ (<DISK_DESC_BLOCK>)
              /STDISK/

54    19.18   <DISK_DESC_BLOCK>::=  "<ORG>  "="  /DSKER2/  <ORG_TYPE>
              /SVORG/" ","
              <RECORD_FORMAT> "," VOL_NAME "=" /VOLERR/ <NAME> /SVVOL/
              ","
              "INT_NAME "=" /DSKER3/ <NAME> /SVINTNM/" ","
              "UNIT "=" /DSKER4/ <NAME> /SVUNIT/" ","
              "SPACE "=" /DSKER5/ ( <SPACE_PARS> ) "

55    19.19   <SPACE_PARS>::=  /DSKER6/  <UNITS> , <QUANTITY> /SVQTY/
              ","
              "<INCREMENT> /SVINCR/" "," "RLSE /SVRLSE/"

56    19.20   <QUANTITY>::= <INTEGER>

57    19.21   <ORG>::=ORG|ORGANIZATION

58    19.22   <INCREMENT>::= <INTEGER>

59    20      <NO_TRKS>::=7|9
```

```
60    21     <PARITY>::= ODD | EVEN

61    22     <DENSITY>::= 200 | 556 | 800 | 1600

62    23     <TAPE_LABEL>::=IBM_STD /SVLAB/ "," /TLABERR/
             INT_NAME "=" <NAME> /SVINTN/
             |ANSI_STD /SVLAB/ "," /TLABERR/ INT_NAME "=" <NAME>
             /SVINTN/
             |NONE /SVLAB/
             | BYPASS /SVLAB/

63    24     <ORG_TYPE>::=ISAM | SEQUENTIAL | SAM | INDEXED_SEQUENTIAL

64    25    <TYPEDSK>::= 2314 | 2311 | 3330 | 2305

65    26     <UNITS>::= TRACKS | CYL /SVUNITS/ | <INTEGER> /SVUNITS/

66    27.1      <RECORD_FORMAT>::=    /RCFER1/    <FIXED_SPEC>    |
             <VARIABLE_SPEC> | <VAR_SPANNED_SPEC> | <UNDEFINED_SPEC>

67    27.2   <FIXED_SPEC>::= FIXED /SVRECF/ "," /RCFER2/ BLOCKSIZE "="
             <BLOCKSIZE>  /SVBLK/  ","  RECORDSIZE  /RCFER3/  "="
             <RECORD_SIZE> /SVRCSZ/

68    27.3     <VARIABLE_SPEC>::=  VARIABLE  /SVRECF/  ","  /RCFER2/
             MAX_BLOCKSIZE   "="   <MAX_BLOCKSIZE>   /SVBLK/   ","
             "MAX_RECORDSIZE "=" /RCFER3/ <MAX_RECORDSIZE> /SVRCSZ/ "

69    27.4    <VAR_SPANNED_SPEC>::=  VAR_SPANNED /SVRECF/ "," /RCFER2/
             MAX_BLOCKSIZE   "="   <MAX_BLOCKSIZE>   /SVBLK/   ","
             "MAX_RECORDSIZE "=" /RCFER3/ <MAX_RECORDSIZE> /SVRCSZ/ "

70    27.5     <UNDEFINED_SPEC>::=  UNDEFINED  /SVRECF/  ","  /RCFER2/
             MAX_BLOCKSIZE "=" <MAX_BLOCKSIZE> /SVBLK/
```

```
71      27.6   <BLOCKSIZE>::= <INTEGER>

72      27.7   <RECORD_SIZE>::=<INTEGER>

73      27.8   <MAX_BLOCKSIZE>::= <INTEGER>

74      27.9   <MAX_RECORDSIZE>::= <INTEGER>

75      28,29   <INTEGER>::=/INTREC/

76      30.4       <ASSERTION_DESC>::=     :     /SVASNM2/     /ASSINIT/
               <ASSERTION_BODY>

77      30.5     <ASSERTION_BODY>::= "SOURCE  /ASSER2/  :  <SQNAME>  ",
               <SQNAME>" * /STSR/ <ENDCHAR> "
               /ASSER3/ TARGET : <TQNAME> ", <TQNAME>" * <ENDCHAR>
               "FUNCTION /FCNERR/ : <NAME> /SVFCN/ <ENDCHAR> " /STTG/ "
               " <ASSERTION_TEXT> /SVTXSTR/ " <ENDCHAR> "

78      30.6   <SQNAME>::= /EACHINT/ <QNAME> "( /EACH01/ <EACH> /SVEACH/
               ) " /SVSR/

79      30.7   <TQNAME>::= /EACHINT/ <QNAME> "( /EACH01/ <EACH> /SVEACH/
               ) " /SVTG/

80      30.8   <EACH>::= /EACHREC/

<ENDCHAR>::= /SEMI/ ; /STMTINC/

<STRING_CONST>::= /CHARSTR/

81      30.9   <ASSERTION_TEXT>::= /TEXTSTR/

82      31     <QNAME>::=/INITONM/ /QNMERR/ <NAME> /MKQNM/  ".  /QNMERR/
               <NAME> /MKQNM/" *

83      32,33,34  <NAME>::=/NAMEREC/

84      35.2   <INTERIM_HDG>::= /FRINTRM/ "DESCRIPTIONS" ":"

85      35.3   <INTERFILE_HDG>::= /FRINTER/ "RELATIONSHIPS" ":"

86      35.4   <ASSERTIONS_HDG>::= /FRASS/ "SECTION" ":"
```

```
87     35.1      <HDGS>::=   INTERFILE   <INTERFILE_HDG>   | ASSERTIONS
                 <ASSERTIONS_HDG> | INTERIM <INTERIM_HDG>
```

In the specification of MODEL in EBNF/WSC in this figure, angle-bracketed names designate syntactic units, square brackets, represented for the SAPG by quotation marks ("), designate optional syntactic units. An asterisk (*) following the optional designation indicates repetition zero or more times. The subroutines to be invoked are indicated between slashes (/.../). Note that subroutine calls are made after the successful recognition of syntactic units up to that point.

The column marked "EBNF Reference number" indicates the statement number of the corresponding EBNF statement of Chapter 3. Where one EBNF statement corresponds to more than one EBNF/WSC statement, a second level number is used.

A representative example from the EBNF specification of Chapter 3 (statement 14) is

```
<FIELD_STMT>::=FIELD(<TYPE>(<INTEGER>[:<INTEGER>]))
```

in the EBNF/WSC specification on line 52, this becomes the following:

```
<FIELD_STMT>::=FIELD /SVFLD/ <FIELD_ATTR> /STFLD/
```

This says that the syntactic unit <FIELD_STMT> (which is referenced in a higher-level production rule) starts with the word FIELD. After successfully recognizing that, the subroutine SVFLD is to be invoked (which turns out to be a subroutine that "encodes" the statement type as being a FIELD type statement). By "encode" we mean compact the external statement type to an internal code or representation. This subroutine is placed after FIELD because the statement type can only be encoded after the FIELD token is scanned. In general, encoding subroutines are inserted after the corresponding token. Then the syntactic unit called <FIELD_ATTR> follows, which is defined in another production rule (it turns out to be the various types of field attributes). Finally, if the foregoing is recognized, the subroutine named STFLD is called (which turns out to be a subroutine that calls the STORE system to put the above information in the associative memory, a concept to be described further). Such a storing subroutine must appear at the end of every MODEL statement in order that it be stored in the associative memory.

Further examples of inserting subroutine calls into the EBNF/WSC are given when each category of subroutines is discussed in later subsections.

The SAP generated by the SAPG according to the EBNF/WSC is supplemented and linked with the routines. The SAP, in turn, accepts statements in MODEL and checks them for syntactical correctness, local semantics, producing a listing of the statements, syntax diagnostics, an encoded stored version of the MODEL statements, and a cross-reference report.

More will be said about inserting subroutines into the EBNF in Sections 4.2.2 and 4.2.3.

4.2.1.2 Adequacy and Limitations of SAPG

On the whole, the SAPG and the EBNF/WSC meta-language is a most useful tool for defining MODEL, for it allows changes to the language to be made relatively quickly while it is undergoing development. The alternative of writing the entire syntax and statement analysis program manually would be even more tedious. Unlike the better-known XPL system [McK 71], this parser-generator produces an ad-hoc program (into the PL/1 language) to parse the specific language described rather than interpret tables. While the SAPG approach has been found to be adequate for future further development, there are nevertheless some limitations to it that need to be mentioned here and which caused minor problems in defining MODEL in this way.

First, one obvious limitation is that SAPG only generates a SAP to analyze on a statement-by-statement basis for local syntactic and semantic correctness, the former directly and the latter via subroutine calls. Since MODEL is non-procedural and each statement is independent, statement-by-statement analysis is appropriate as a first pass. Any global analysis must be done by the Processor implementer. In fact, global analysis of the MODEL specification for implicit statement inter-relationships is a major task of the MODEL Processor in a later phase (Section 4.3).

A restriction of the SAPG, as already noted in footnote 3, is that it can only generate a "bounded-context" or "LR-k" SAP where k=1. This means that whenever there is an alternative in the grammar, the path to be taken is to be determinable by the next token, as exemplified below. This restriction is due to the fact that the SAPG does not generate a run-time stack of syntactic units and can only "backtrack" one token.

To illustrate, consider the following three production rules that one might want to express:

```
<X>::=<Y>|<Z>
<Y>::=A B...
<Z>::=A C ...
```

In order to recognize the string A C ... as an <X>, a parser capable of backtracking could first try the <Y> alternative, where the A would match but the C would not. At this point, the parser would have to backtrack and try the <Z> alternative where a match of A C ... would be found. The SAPG does not have such a backtracking capability, and the alternative to be taken must therefore always be determinable by the next token. In order to conform to the "LR-1" form that the SAPG expects, the above set of production rules could be re-written by "factoring out" the "A" as follows:

```
<X>::=A <Y-OR-Z>
<Y-OR-Z>::=<Y>|<Z>
<Y>::= B ...
<Z>::= C ...
```

The "LR-1" restriction was the reason for restructuring some productions in the EBNF/WSC of MODEL. For example, in statement 11, the following production rule

<DATA_DESC_STMT>::=IS /BADDDS/ <DATA_DESCRIPTION> <ENDCHAR>

starts with the terminal " IS ..." rather than "<NAME> IS ..." This occurs because the <NAME> was already factored out to a higher production rule, because <NAME> is the first token of other types of statements as well. This restriction, however, is still adequate for languages such as MODEL, because the EBNF/WSC for MODEL can and was written in "LR-1" form by "factoring out" common syntactic units to higher levels in the grammar tree and using keywords for unique identification of path. Thus, although this restriction makes writing the grammar somewhat awkward, it is by no means a serious impediment to the use of EBNF/WSC and SAPG for defining MODEL.

Another feature that warrants improvement is the need to insert error-stacking routines, which require a tedious effort on the part of the language-definer. While the method of writing error-message stacking routines is clear (and described later), it could have been avoided altogether with a straight-forward extension to the SAPG system. There is no reason in principle that the SAPG system could not have been designed and implemented in such a way that it itself generate error messages for missing or incorrect syntactic units based on the names of the syntactic units in the EBNF/WSC.

Finally, the SAPG never had a standard facility for storing source language statements. The processor writer was responsible for his own ad-hoc statement storage procedures. This deficiency has been relieved during the course of this research by augmenting the SAPG system with a general-purpose mechanism for storing and later retrieving source language strings in a simulated associative memory. Such a facility (described in Section 4.2.4) was implemented as part of this project as a general tool that could be applicable to other language processors.

In conclusion, while the current status of the SAPG system has some minor drawbacks, it by all means has been an adequate tool for defining MODEL now or for future changes, and certainly enables faster development changes than does writing a SAP manually.

4.2.1.3 How the SAPG Produces the SAP

As indicated, the SAPG produces the SAP from the specification of the object language in the EBNF/WSC meta-language. The design of the SAPG is documented in [RAM 73] and its implementation in [FRE 72]. For completeness, the technique of the SAPG is abstracted here, but the reader desiring greater understanding on the operation of the SAPG should refer to the above sources for detailed flowcharts and documentation.

The SAPG is a small compiler in itself in that it processes a specification in the language EBNF/WSC and produces a program (SAP). It performs this in three passes over the set of productions.

In pass 1, each production is scanned, and its components are encoded into a set of tables. Non-terminal symbols appearing on the left-hand-side of a production (new production names) are put into a symbol table, while non-terminals appearing on the right-hand-side of a production are put into a work table. Terminal symbols in a production are put into a terminal symbol table. Subroutine calls are put into yet another table.

In pass 2, the symbolic references in the work table (i.e. non-terminals on the right-hand-side of the original production) are resolved. Pass 2 checks that each right-hand-side non-terminal symbol in the work table is defined, and links it to the corresponding entry in the symbol table. Undefined non-terminals as well as circularly-defined non-terminals can be detected in these table searches.

Pass 3 of the SAPG is the code-generation phase that produces the SAP in PL/1. It is only entered if no errors were encountered in the previous phases. For each EBNF/WSC production, a PL/1 procedure is generated. Each one returns a bit: 1 if the recognition was successful; 0 if it was unsuccessful. The exclusive nature of EBNF production rules and alternatives is effected by generating nested PL/1 IF-THEN-ELSE statements. Repetition zero or more times is effected by generating a GO TO to the statement testing for recognition. Subroutine names embedded in the EBNF/WSC get a CALL generated for them in place. Calls to other subroutines not explicit in the EBNF/WSC are also generated. These include "housekeeping" subroutines of the SAPG and calls to LEX, a subroutine to scan and return the next token in the object language.

To illustrate the code that the SAPG generates, consider the following representative production rule in the EBNF/WSC and the PL/1 code that corresponds:

```
<FIELD_STMT>::=FIELD /SVFLD/ <FIELD_ATTR> /STFLD/
```

The PL/1 code that is generated for it by the third pass of the SAPG would be the following:

```
FIELD_STMT: PROCEDURE RETURNS(BIT(1));
CALL $MARK;
CALL LEX;
IF LEXBUFF='FIELD' THEN DO;
CALL LEXENAB;
CALL $POPF;
CALL SVFLD;
IF FIELD_ATTR THEN DO;
IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;
CALL STFLD;
CALL $SUCCES; RETURN('1'B);END;
ELSE DO; CALL $SUCCES; RETURN('1'B); END;
END;
ELSE DO; CALL $FAIL; RETURN('0'B); END;
END FIELD_STMT;
```

The above code generated by the SAPG would become one procedure in the SAP. Note that the names that the language definer uses in the production rule are preserved in the generated SAP code. The subroutines beginning with dollar signs ($) are "housekeeping" routines that are internal to the mechanisms of SAPG-generated code. Their detailed logic is documented in [FRE 72] and really do not concern the language definer.

4.2.2 Supporting Subroutines for EBNF of MODEL

A refined system flowchart of the SAPG and SAP showing the types of supporting routines appears in Figure 4.6. The manually-written syntactical supporting routines are of one of several types:

(1) a lexical analyzer which returns tokens of syntactic units to the SAP for analysis,

(2) statement semantics checking routines;

(3) error message handling routines;

(4) encoding routines to compact information for further efficient processing; and

(5) statement storage routines.

The cross-reference report produced during this phase is generated by a manually-written program (XREF) and is described in Section 4.2.6.

A discussion on how to decide where to insert subroutines as well as a tabular summary of all routines used will appear in Section 4.2.3 after a breakdown of the different categories here.

4.2.2.1 The Lexical Analyzer

The purpose of the lexical analyzer is to scan for syntactic units or "tokens", using such delimeters as blanks and certain punctuation marks, and to return them to the Syntax Analysis Program (SAP) for syntactic checking. The automatically-generated SAP calls upon the lexical analyzer (LEX) whenever it needs the next token. The

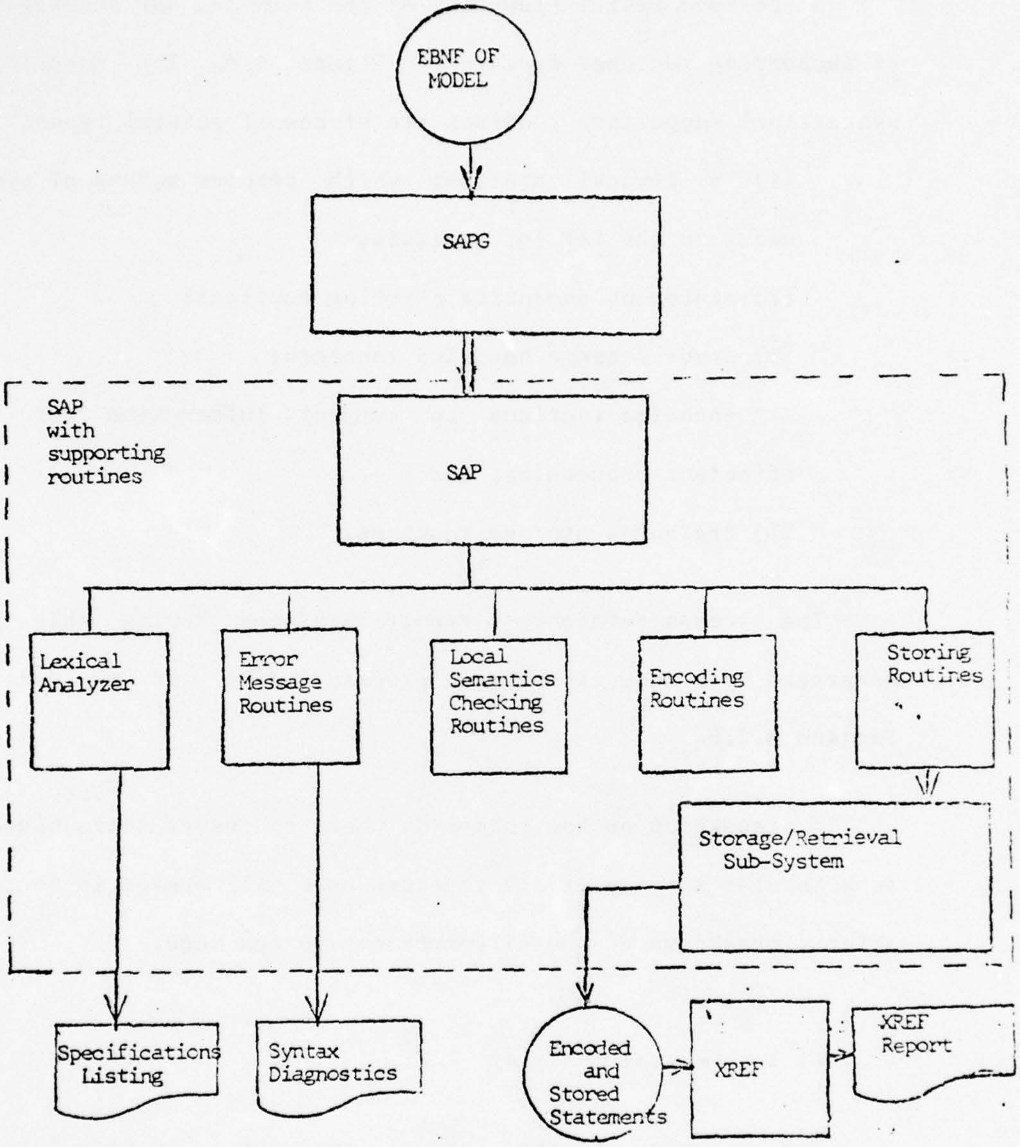


Figure 4.6

More Detailed View Of SAPG and SAP With Supporting Subroutines lexical analyzer implemented here is based on the finite state machine concept [CON63]. Each state of the machine corresponds to a condition in the lexical processing of a character string. At each state, a character is read, an action is taken based on the character read (such as concatenating the current character to previous ones or returning the entire token to the SAP), and the machine changes to a new state. The character classes for the MODEL Language, for the purposes of lexical analysis, appear in Table 4.1. These classes divide the entire character set into categories such as illegal characters, delimeters, "normal" characters, etc. The state transition matrix for the MODEL language appears in Table 4.2. The rows of the matrix represent the character classes of the previous character, while the columns represent those of the current character. The entries in the matrix indicate the action to be taken and the next state. The action taken in each state is summarized in Table 4.3. The actions involve such steps as concatenating of a character, ignoring a character, detecting an illegal character, returning a complete token to the SAP, etc., and setting a "next state".

4.2.2.2 Statement Semantic Analysis

Some of the semantics of the specification statements can be checked during the syntax analysis phase. Such routines can check that a range or condition on a syntactic unit is locally correct. These routines do not and cannot check the overall consistency, completeness, or correctness of the logic of the MODEL specification, a task which is performed by a later phase of the Processor. An

Class	Character Set	Explanation
0	A,B,...,Y,Z,_.#,@	Characters in names
1	(space)	Delimeter
2	0,1,2,...,9	Numerals
3	(+&);,%:'"	Delimeters in various contexts
4	.	Qualifier symbol
5	<	Delim in logical expr.
6	|	"OR" symbol
7	*	Mult. Or comment if with "/*"
8	~	"NOT" symbol
9	-	minus symbol
10	/	Division or comment if with "/*"
11	>	Delim in logical expression
12	=	Delim for keywords & log. Expr.
13	all others	Illegal

Table 4.1

Character Classes for MODEL Language

Character Class (next)	0	1	2	3	4	5	6	7	8	9	10	11	12	13
(current)														
0	1	2	1	2	2	2	2	2	2	2	2	2	2	7
1	1	3	1	5	1	1	1	1	1	1	1	1	1	7
2	1	2	1	2	1	2	2	2	2	2	2	2	2	7
3	2	2	2	2	2	2	2	2	2	2	2	2	2	7
4	2	2	1	2	2	2	2	2	2	2	2	2	2	7
5	2	2	2	2	2	2	2	2	2	2	2	2	1	7
6	2	2	2	2	2	2	1	2	2	2	2	2	2	7
7	2	2	2	2	2	2	2	1	2	2	2	2	2	7
8	2	2	2	2	2	1	2	2	2	2	2	1	1	7
9	2	2	2	2	2	2	2	2	2	2	2	1	2	7
10	2	2	2	2	2	2	2	6	2	2	2	2	2	7
11	2	2	2	2	2	2	2	2	2	2	2	2	1	7
12	2	2	2	2	2	2	2	2	2	2	2	2	2	7
13	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Table 4.2

State Transition Matrix for MODEL Lexical Analyzer

Action 1: Concatenate next character to current token

Action 2: End word with next character

Action 3: Skips blanks sequence

Action 4: Reserved (never taken)

Action 5: Scan forward one character and save as token

Action 6: Comment bracket; scan to end of comment

Action 7: Illegal character(s); print error message

Table 4.3

Lexical Analysis Actions

example of a local semantics checking routine is one which checks the range of a numeric computation. For instance, if a group is said to occur n to m times, a subroutine exists to check the 0 <= n < m < 32768. These manually-written routines are invoked automatically by the SAP by virtue of their specification in the EBNF/WSC of the MODEL language for the SAPC.

4.2.2.3 Error Message Stacking Routines

These are subroutines which stack error diagnostics for the SAP to print out upon recognition of a syntactically-incorrect user statement. Upon reaching incorrect syntactic units, the automatically generated SAP does not print its own messages, but expects the corresponding diagnostics to be on an "error stack." For this purpose, subroutines have to be written to give a MODEL user effective information when his statements have been incorrectly composed. Specifically, an error message has to be stacked for each expected terminal symbol in the MODEL language in case the token is missing or incorrect. If the expected token is found, the SAP simply pops the corresponding error message and continues; if the expected token is missing or incorrect, the SAP pops the corresponding error message, prints the statement number and message, scans for the end of the statement delimeter (";"), and continues. The routines that stack such error message codes are the ones ending the letters "ER" or "ERR" as found in the EBNF/WSC (e.g. RECERR). Each routine's syntax error message code pinpoints the token that is incorrect, missing, unexpected, or misspelled.

One product of the syntax analysis phase is the Error Diagnostics Report containing the messages. Each message gives the diagnostics provided by the error routine and provides the exact location of the error so that it can be corrected and resubmitted by the user easily. If no syntax errors are found during the syntax analysis phase, a message is sent that "NO ERRORS OR WARNINGS DETECTED", and the Processor proceeds to the next phase. But if error diagnostics were produced, a flag is set to disable continuation of analysis and design beyond the syntax checking phase.

4.2.2.4 Encoding User Statements

These supporting routines encode some of the MODEL specification into an internal representation. Although all of the names provided by the user specification are kept intact in internal form for use by the object program, many of the descriptions and attributes are encoded for more compact and efficient processing later. For example, the description in a FIELD statement enters an internal table where the type of field is encoded (0 for character, 1 for binary, 2 for numeric, etc.), and the field length type is encoded 0 for fixed length, 1 for variable length). One encoding routine is written for each such statement type. Each routine is invoked automatically after recognition of the syntactic unit by the SAP. The invocation is automatically generated as part of the SAP by the SAPG by virtue of its specification in the EBNF/WSC. The internal format of the tables is given in the next section in conjunction with the discussion of the internal associative storage of the MODEL statements.

4.2.2.5 Statement Storage Routines

These routines collect the strings of names and other vital information in the MODEL statements, and pass them to the STORE system, which is a sub-system in itself to store the statements for later processing. Such storage-invoking routines are called at the end of scanning each MODEL statement, and are the ones that begin with the letters "ST" as found in the EBNF/WSC back in Figure 4.5 (e.g. STFLD, STREC, etc.). The storage subsystem described below (STORE), which is called by these routines, stores the MODEL statements in a simulated associative memory that facilitates later retrieval.

At the end of the syntax pass, we have the entire set of MODEL statements stored in a convenient storage system for further analysis. The storing subroutines which invoke the use of the STORE system act as an interface between the automatically generated SAP and the storage system presented below. The storage system is an extension to the capabilities of the SAPG since it is general purpose in nature and is independent of the nature of the language specified, and could be used for processing other languages.

4.2.3 Experience with and Use of EBNF/WSC and SAPG

The use of EBNF/WSC to describe the MODEL language and its processing by the SAPG has been successfully implemented during this research. Since this is only the second application of the SAPG to implement the statement analysis phase of a processor, its u^e is outlined and exemplified here as a guide for further future development. Examples from the EBNF/WSC of MODEL presented in Section 4.2.1 are explained here for illustration.

To use the SAPG, one first writes the EBNF without subroutine calls representing the grammar of the language in "LR-1" form as shown in Section 4.2.1. One then proceeds to insert or embed subroutines within the EBNF. The sub-sections 4.2.2.1 through 4.2.2.5 described the different types of routines that need to be written and included in the EBNF/WSC when using the SAPG system. This section elaborates on the methods, reasons for, and examples of their inclusion. The rationale behind their design stems from the way the SAPG was implemented and works, as documented in [RAM 73, FRE 72].

The need to include error message stacking routines stems from the fact that the automatically-generated SAP does not print its own messages, but expects the corresponding diagnostics to be placed on an error stack by routines provided by the language-definer. Therefore, preceding every mandatory syntactic terminal symbol in the EBNF/WSC grammar of MODEL , a routine must be included to stack an error message in case the token is missing or incorrect. The SAP pops these messages upon recognizing each corresponding syntactic unit and prints the error message when the syntactic unit reached does not match that specified in the MODEL grammar written in EBNF/WSC. For example, consider first the latter part of EBNF statement 14 of Chapter 3:

```
<FIELD_STMT>::=FIELD(<TYPE>(<INTEGER>[:<INTEGER>]))
```

Compare now parts of the corresponding two production rules in the EBNF/WSC of MODEL to illustrate the use of error-stacking routines (statements 28 and 33):

```
<FIELD_ATTR>::=/FLDERR1/ (<TYPE> /SVFDTP/ (<MIN_LENGTH> ...
<TYPE>::=CHAR|BIN|...
```

The subroutine call to FLDERR1 has been inserted at the beginning of the first production, because it must anticipate each potential missing or incorrect terminal symbol to come, by stacking error messages. The subroutine itself stacks one error message for each anticipated non-optional terminal symbol, including one for the "(", one for "<TYPE>", one for the second "(", etc. Notice that for non-terminals such as <TYPE>, it makes no difference whether FLDERR1 pushes the error message for the eventual anticipated terminal, or whether the <TYPE> production below does it, as long as one error message is stacked for each anticipated terminal in the grammar tree. Thus, if the MODEL language is ever extended or changed with new syntactic units in the grammar, subroutines would have to be included in the EBNF/WSC preceding each terminal symbol and must be written. The subroutines must stack one error message for each terminal symbol appearing in the grammar.

The above example also illustrates the use of encoding routines, which are inserted into the EBNF/WSC at the discretion of the language definer whenever there is a desire to encode and save a syntactic unit for more compact or efficient processing later. The subroutine named SVFDTP ("Save Field Type"), for example, takes the field type just recognized and encodes BINARY as 0, CHAR as 1, etc. This information is passed to the STORE system in a subsequent subroutine (storage subroutines are discussed below). Such routines in the EBNF/WSC are placed after the corresponding syntactic unit, so that the syntactic unit will have already been recognized by the SAP.

Routines for checking local statement semantics are optional and at the discretion of the language definer. They are inserted in the EBNF/WSC specification whenever the language definer wishes to check that a range or condition is locally correct, something not possible to specify through syntax alone. To illustrate, the following production rule of the EBNF/WSC statement 24

<OCC_END>::=)|:/MAXERR/ <MAXOCC> /SVMXOC/ /CKMNMX/

describes how to write the end of the number of occurrences of an item in MODEL (after the minimum was already given). The second alternative is taken whenever there is a maximum and a minimum, and consists of a colon followed by the maximum number of occurrences <MAXOCC>. The subroutine CKMNMX ("Check Minimum & Maximum") is inserted here to check that 0 <=minimum <= maximum < 32767 (=2**15, the highest allowed). Thus CKMNMX is an example of performing local semantic analysis. Such routines appear after the corresponding syntactic units so that they will have already been recognized by the SAP. The other two subroutines in the above example, MAXERR and SVMXOC, deal with error-stacking and encoding, respectively.

There is also a need to include a routine in the EBNF/WSC at the end of each production describing a type of MODEL statement. Such routines call the store system for storing the collected tokens in the associative memory. These routines act as an interface between the automatically generated SAP and the string storage system by calling the store system with the necessary parameters. For example, first consider the EBNF production rule of statement 3:

<MODULE_NAME_STMT>::=MODULE: <NAME>

Now consider the corresponding production rule on statement 3 of the EBNF/WSC describing the remainder of the module name statement (the keyword MODULE itself appeared previous to this):

<MODULE_NAME_STMT>::=/MODUL1/ : /MODUL2/ <NAME> /STMOD/ <ENDCHAR>

The production rule consists of a colon, followed by the name of the module, followed by the ending character ";". The subroutine STMOD ("Store Module Name") takes the pertinent collected tokens, in this case just the name of the module and the statement type, and passes them to the STORE system which it calls. The storage system itself is described later in this section. In any future modifications to MODEL, the language definer would want to include a routine to invoke the store system at the end of each statement described in the EBNF/WSC.

The MODUL1 and MODUL2 subroutines are error stacking routines that stack error messages for anticipated terminal symbols as described earlier (here an error message is stacked for the colon and the name in case they are illegal or missing).

The lexical analyzer described in Section 4.2.2.1 does not have to be indicated in the EBNF/WSC, but has to be written to scan for tokens. Its need stems from the fact that the SAPG only generates a call to the lexical analyzer, which is expected to find the next unit. This is due to the fact that the rules by which lexical units are found are based on combinations of delimeters or punctuation marks, which may vary from language to language. Although the lexical analyzer described here is specific to scanning for tokens in the MODEL language, it has general applicability because it is table-driven. If the MODEL language is ever extended or changed with new punctuation marks or delimeters being introduced, it would simply require the new characters to be placed in an appropriate class in the character class table (Table 4.1) and to designate in the state transition matrix (Table 4.2) the proper action to be taken upon reaching that character.

Finally, there are just a few "housekeeping" type subroutines which need not be written by the language definer because they are provided by the SAPG , but which need to be included in the EBNF/WSC. The very first production in the EBNF/WSC of MODEL illustrates two of the subroutines. It differs from all of the other production rules and perhaps needs clarification:

```
<MODEL_SPECIFICATION>::=[<MODEL_BODY_STMT>        /CLRERRF/]*
/STMT_FL/ <MODEL_SPECIFICATION>
```

<MODEL_SPECIFICATION> is the goal symbol which is defined as zero or more <MODEL_BODY_STMTS>, each of which is further defined as one of the statement types of MODEL. After each such statement is recognized, this production causes further attempts to be made to recognize more statements (via the asterisk repetition operator). The subroutine CLRERRF ("Clear Error Flag") is a SAPG provided housekeeping routine which resets a flag indicating presence of an error. SAPG requires that it be called at the end of each statement (see [FRE 72]). Continuing with the production explanation, if a statement which matches none of the <MODEL_BODY_STMT> types is encountered, the production indicates to branch to the subroutine STMT_FL ("Statement Fail"). This subroutine scans the text for the statement delimeter ";" to begin scanning the next statement. Finally, the last non_terminal <MODEL_SPECIFICATION> causes the SAP to attempt recognizing another statement by calling the same production recursively.

The two other housekeeping routines of SAPG which are applicable to languages other than MODEL appear elsewhere. One is ENDINP ("end input", on statement 2), which is called at the end of the input text. The other is STMTINC ("Statement Increment", on statement 80), a subroutine called after recognizing each statement delimeter ";" in order to increment the statement number.

The subroutine names used in the specification of MODEL were shown at the bottom of the EBNF/WSC listing. They can be classified into one of the following five types of subroutines: error message stacking routines, encoding/saving routines, storing routines, semantics checking routines, and housekeeping routines. The tables provide an alphabetical listing of the routines within each category. In the case of error message routines, the error codes and their meanings are shown. For the other types of routines, their name and tasks are shown.

Storing Routines

(inserted at the end of each type of statement of the EBNF/WSC in order to call STORE to put the statement in the associative memory)

NAME	STMT	WHAT IT STORES
STCARD	38	Stores CARD statement
STDISK	53	Stores DISK statement
STFILE	13	Stores FILE statement
STFLD	27	Stores FIELD statement
STGRP	21	Stores GROUP statement
STINT	36	Stores INTERIM statement
STMOD	3	Stores MODULE statement
STPNCH	40	Stores PUNCH statement
STPRNT	39	Stores PRINTER statement
STREC	20	Stores RECORD statement
STRPT	34	Stores REPORT statement
STRPTN	35	Stores REPORT-ENTRY statement
STSR	77	Stores SOURCE portion of assertion
STSRC	4	Stores SOURCE FILES statement
STTAPE	45	Stores TAPE statement
STTAR	7	Stores TARGET files statement
STTERM	41	Stores TERM statement
STTG	77	Stores TARGET portion of assertion

Semantics Checking Routine

(inserted in the EBNF/WSC after the token(s) to be checked or for other action)

NAME	STMT	WHAT IT DOES
ASSINIT	77	Initializes number of sources/targets to assertion
CKMNMX	24	Checks proper range for minimum and maximum
EACHINT	78	Initializes flag for FOREACH existence
EACHREC	80	Recognizes FOREACH phrase
FRASS	86	Prints frame before first assertion
FRINTER	85	Prints frame before interfile relationship
FRINTRM	84	Prints frame before interims
GETLIB	17	Gets input from library
INITQNM	82	Initializes number components to qualified name
INITSFL	4	Initializes source file list
INITTFL	7	Initializes target file list
INTREC	75	Recognizes integers
MEMINIT	20,21	Initializes number of members of record or group
MKQNM	82	Concatenates qualified name components
NAMEREC	83	Name recognizer; checks not keywords

"Housekeeping" Routines

(inserted in the EBNF/WSC in order to perform services provided by the SAPG

NAME	STMT	WHAT IT DOES
CLRERRF	1	Clears "error" flag after every statement to indicate no syntax errors yet in next statement
STMT_FL	1	Scans for end of statement delimeters when unrecognizable statement encountered
ENDINP	2	Executed upon end-of-file to print last line and wrap-up
STMTINC	80	Increments the statement number; called at end of each statement

Error Message Stacking Routines

(Inserted in the EBNF/WSC before each anticipated terminal symbol).

NAME	CODE	ERROR MESSAGES
ASSER1	ASSER1	SECTION keyword missing in heading
	ASSER2	Colon missing in assertion heading
ASSER2	ASSER3	Colon missing after keyword SOURCE
ASSER3	ASSER4	TARGET keyword missing in assertion
	ASSER5	Colon missing after keyword TARGET
BADDDS	BADDDS	Unrecognizable data description
BADDDS2	BADDS2	Invalid keyword beginning data description or assertion
DSKER1	DISK01	Left paren missing in DISK statement
	DISK02	Right paren missing in DISK statement
DSKER2	DISK03	Organization type missing or illegal in DISK statement
DSKER3	DISK04	Internal name missing or illegal in DISK statement
DSKER4	DISK05	Type disk missing or illegal in DISK statement
DSKER5	DISK06	Left paren missing in SPACE spec in DISK statement
	DISK07	Right paren missing in SPACE spec in DISK statement
DSKER6	DISK08	Units missing or illegal in DISK statement SPACE spec

	DISK09	Comma missing after units in DISK statement SPACE spec
	DISK10	Quantity missing or illegal in DISK statement SPACE spec
EACH01	EACH01	FOREACH name missing or illegal in assertion
FCNERR	FCNER1	Colon missing after FUNCTION keyword
	FCNER2	FUNCTION name missing
	FCNER3	Semi-colon missing after function name
FILERR1	FILE01	Left paren missing in FILE or REPORT statement
	FILE02	Right paren missing in FILE or REPORT statement
FILERR2	FILE03	Keyword missing in FILE or REPORT statement
FILERR3	FILE04	Record name missing or illegal in FILE or REPORT statement
FILERR4	FILE05	Character code missing or illegal in FILE or REPORT
FILERR5	FILE06	Medium name missing or illegal in FILE or REPORT statement
FILERR6	FILE07	Keyname missing in FILE or REPORT statement
FLDERR1	FLD01	Left paren missing in FIELD statement
	FLD02	Field type missing or illegal in FIELD statement
	FLD03	Left paren missing before length in FIELD statement

	FLD04	Length missing or illegal in FIELD statement
	FLD05	Right paren missing in FIELD statement
	FLD06	Right paren missing in FIELD statement
FLDERR2	FLD07	Maximum length missing or illegal in variable length in FIELD statement
GRPERR	GRP01	Left paren missing in GROUP statement
	GRP02	Right paren missing in GROUP statement
INTER1	INT01	Colon missing in INTERIM heading
INTER2	INT02	Keyword INTERIM missing in INTERIM statement
	INT03	Left paren missing in INTERIM statement
	INT04	Type missing or illegal in INTERIM statement
	INT05	Right paren missing in INTERIM statement
ITEM01	ITEM01	Name missing or illegal in item list
LIBERR	LIB01	Left paren missing in library call
	LIB02	Library name missing or illegal in FILE statement
	LIB03	Right paren missing in library call
MAXERR	MAXER1	Maximum number of occurences of item missing or illegal
	MAXER2	Right paren missing after maximum number
MINERR	MINER1	Number of occurences of item missing or illegal
	MINER2	Colon or right paren missing

MODUL1	MODUL1	Colon missing after keyword MODULE
MODUL2	MODUL2	Name missing or illegal in MODULE statement
PARERR	PARERR	Tape spec parameter missing or illegal
QNMERR	QNMERR	Qualified name illegal
RCFER1	RECF01	Record format missing or illegal
RCFER2	RECF02	BLOCKSIZE keyword missing in record format specification
	RECF03	Blocksize value missing or illegal in record format spec
RCFER3	RECF04	Record size value missing or illegal in record format spec
RECERR	RECD01	Left paren missing in RECORD statement
	RECD02	Right paren missing in RECORD statement
RPTERR	RPT01	Left paren missing in REPORT statement
	RPT02	Keyword REPORT_ENTRY missing
	RPT03	Report entry name missing
	RPT04	Right paren missing in REPORT statement
RPTNER	RPTN01	Left paren missing in REPORT_ENTRY statement
	RPTN02	Right paren missing in REPORT_ENTRY statement
SEMI	SEMI	Semi-colon missing at end of statement
SRCFL1	SRCFL1	Colon missing after keyword SOURCE [FILES]
SRCFL2	SRCFL2	Name missing or illegal in source file list
TAPERR	TAPE01	Left paren missing in TAPE statement
	TAPE02	Right paren missing in TAPE statement

TARFL1	TARFL1	Colon missing after keyword TARGET
TARFL2	TARFL2	Name missing or illegal in TARGET file list
TLABERR	TLAB01	Keyword INT_NAME missing in tape label description
	TLAB02	Internal name missing or illegal in tape label description
TRMERR	TRMER1	Left paren missing in TERM description
	TRMER2	Right paren missing in TERM description
UNRECS	UNRECS	Unrecognizable statement
VOLERR	VOLER1	VOL_NAME keyword missing
	VOLER2	Volume name missing or illegal

Encoding/Saving Routines

NAME	STMT	WHAT IT DOES
SVASNM	76	Saves assertion name in assertion storage entry
SVBLK	62,70	Saves blocksize in disk/tape storage entry
SVCC	16	Encodes character code
		0=EBCDIC, 1=BCD, 2=ASCII
SVCOL	30	Saves column number in field storage entry
SVDDNM	9	Saves data description statement name
SVDEN	50	Saves density in tape storage entry;
SVDSK	53	Encodes disk statement type as disk
SVEACH	78,79	Saves FOREACH name in assertion storage entry
SVFCN	77	Saves function name in assertion storage entry
SVFDTP	28	Encodes field type
		0=character; 1=binary; 2=decimal;3=numeric;
SVFILE	13	Encodes file statement type as FILE
SVFLD	27	Encodes field statement type as FLD
SVINCR	55	Saves increment in disk storage entry
SVINNM	36	Encodes INTERIM statement type as INTR
SVINTNM	47	Saves internal label name in disk storage entry
SVINTN	62	Saves internal label name in tape storage entry
SVKEY	16	Saves key field in file storage entry
SVLAB	62	Encodes label type in tape statement;
		0=none, 1=IBM_STD, 2=ANSI_STD, 3=BYPASS
SVLBNM	17	Saves library name in file storage entry
SVLINE	29	Saves line number in field storage entry

SVMEM 23 Saves member name in record/group storage entry

SVMNFLN 28 Saves minimum field length in FIELD statement

SVMNOC 23 Saves minimum number of occurrences in record or group storage entry

SVMXFLN 28 Saves maximum field length in FIELD statement

SVMXOC 24 Saves maximum number of occurrences in record or group storage entry

SVORG 54 Encodes organization type in DISK statement
S=sequential; I= ISAM ;

SVPAR 49 Saves parity in tape statement

SVQTY 55 Saves track quantity in disk storage entry

SVRCNM 15 Saves record name in file description storage entry

SVRCSZ 67,70 Saves record size in tape/disk storage entry

SVRENM 34 Saves report entry name in report storage entry

SVRECF 68,70 Encodes record format on tape/disk storage entry;
0=FIXED, 1=FIXED BLOCK, 2=VARIABLE
3=VARIABLE BLOCKED, 4=VARIABLE SPANNED,
5=VARIABLE SPANNED BLOCKED, 6=UNDEFINED

SVRLSE 55 Encodes space release indicator in disk storage entry
1=release; 0=no release;

SVRPT 34 Encodes report statement type as REPT storage entry

SVSR 78 Saves source name to assertion in ASSR storage entry

SVSRC 6 Saves source file name in source storage entry

SVSTFL 52 Saves start file in TAPE storage entry

SVSTNM 16 Saves storage name in FILE storage entry

SVTAB 33 Sets tabulated indicator in group storage entry

SVTAPE	45	Encodes tape statement type as TAPE
SVTAR	8	Saves target file name in target storage entry
SVTERM	41	Encodes terminal statement type as TERM
SVTG	79	Saves target name to assertion in ASTG storage entry
SVTMUN	42	Saves tape unit number of tape storage entry
SVTRK	48	Saves number of tracks in TAPE statement
SVTRMNM	43	Saves terminal name
SVTXSTR	77	Saves text of assertion in assertion storage entry (SVTX)
SVUNIT	54	Encodes disk units in DISK storage entry
SVUNITS	65	Saves space units in DISK storage entry
SVVOL	46,54	Saves volume name in disk/tape storage entry

A final note about the SAP is that it lists all the source statements in the order that they are submitted, and numbers the statements in the listing. There is no real reason to reorder the MODEL statements in the listing because the non-procedural nature of MODEL implies no special significance to any order. As seen in an example of a listing later, the SAP also prints various headings for readability.

4.2.4 The String Storage and Retrieval Sub-System

4.2.4.1 Introduction

In order to augment the SAPG system with a general-purpose mechanism for storing and later retrieving source language strings, the following system has been implemented. Basically, it consists of two main routines:

(1) STORE for storing source language strings collected during syntax analysis; and

(2) RETRIEVE for accessing previously stored source language strings, based on a variety of "keys."

The STORE procedure accepts strings which are formed by the subroutines called during syntax analysis. It stores the strings in memory which we call "storage entries" while building "directory entries" in a directory of certain names designated as keys. By building a directory, the strings are stored "associatively" in the sense that statements can later be retrieved based on their content. This capability is crucial to a "non-procedural" language processor, since the statements can be input in any order.

4.2.4.2 The Directory and Storage Structure

The storage entries (the strings to be stored) consist of two parts:

(1) the key names to be entered in the directory which include the names the user provided in the MODEL statements for naming data, assertions, etc. These are the names by which we may want to retrieve

information later.

(2) auxiliary data from the source language strings including the encoded information in table form. This information is not used as the basis of retrievals.

Each storage entry will contain information from a given MODEL statement. They will appear in memory in the order in which they are processed.

The directory consists of an entry for each key name. Each directory entry points to the first storage entry containing that key name. A linked-list is then maintained from the first storage entry with that key name to other storage entries containing the same key name. A "branch and bound" binary tree structure was chosen for the directory itself to make tree modifications and searching for key names efficient. That is the first key name entered in the directory becomes the root of the directory tree; the next key is entered "above" or "below" it in the tree by lexicographic order; etc.

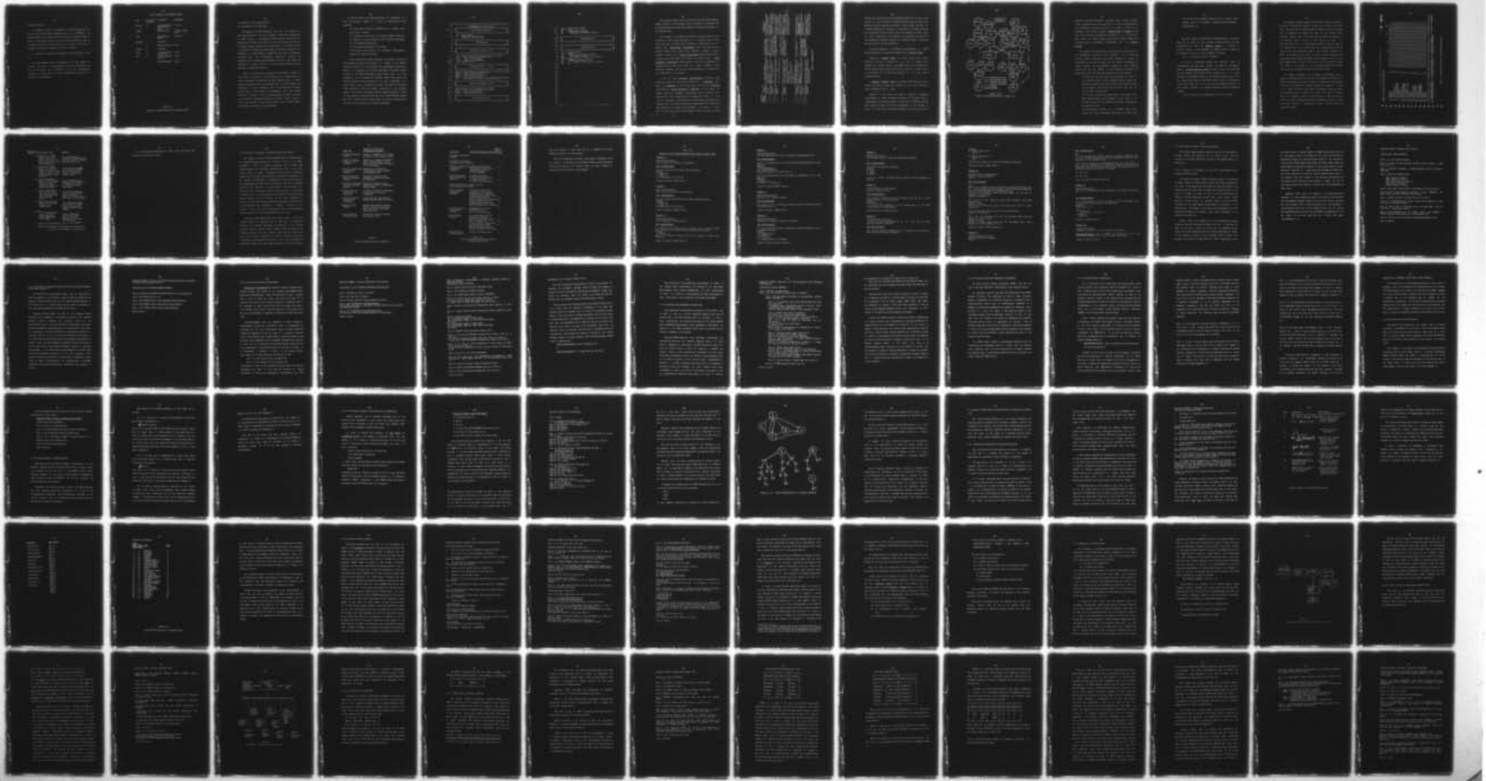
Each directory entry has the following form:

Key name	ptr-to-first	up-pointer	down-pointer

AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416

UNCLASSIFIED 76-05 NL

3 OF 6
AD
A032638



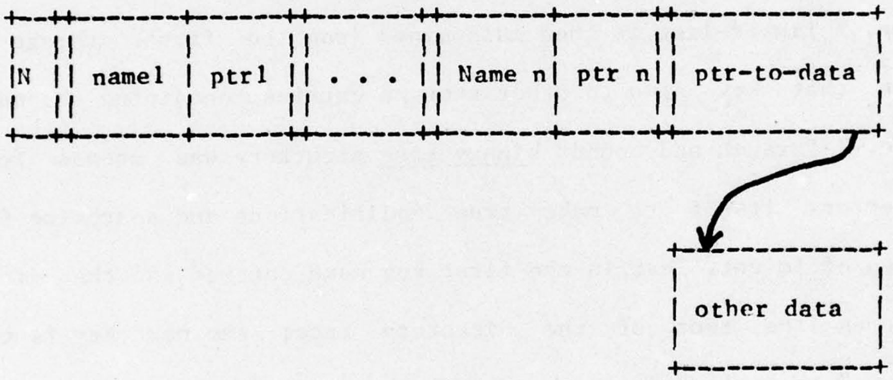
where

"keyname" is a string of (up to) 10 characters (padded with blanks)

"ptr-to-first" is a pointer to the first storage entry containing the "key name".

"up-pointer" and "down-pointer" are pointers to other directory entries, whose key names are up or down, respectively, in the lexicographic sense.

Each storage entry has the following form:



where

n is the number of key names in the storage entry string.

Name (i=1 to n) is a pointer to the next storage entry with the same key name.

Ptr (i=1 to n) is a pointer to the next storage entry with the same key name.

Ptr-to-data is a pointer to auxiliary data from the source language statement.

Figure 4.7 depicts an example of three storage entries and a directory consisting of only three entries, X, Y, and Z, where Y is the directory tree apex. Such a structure was partially motivated by similar ideas in the "multi-list" file organization [PRY66].

4.2.4.3 Storage Entries Format and Tables for MODEL Statements

The STORE mechanism, described in the next section, is called by SAP's storing subroutines to store the MODEL statements for retrieval (by RETRIEVE) in the later phases. For each type of MODEL statement, the key names in it are stored in its storage entry. The non-key information in the MODEL statement (information which is not used to specify retrievals) is kept in description tables, which are connected (by STORE) to the corresponding storage entries as was shown above. Table 4.4 summarizes the internal format of the storage entries and the corresponding description tables for each type of MODEL statement. The left column in this table depicts each prototype MODEL statement. The first name in each entry is the name of the statement being stored. The middle column shows the information appearing in the corresponding storage entry (with the pointers omitted due to lack of space). The right column shows the additional encoded information, if any, from the statement. The key names beginning with a dollar sign ($) in the storage entries are not user-provided, but are inserted by the system for its own information. The last name in each storage entry, for example, identifies the type of statement, while the name beginning with a "$P" identifies the parent file in which a data item appears.

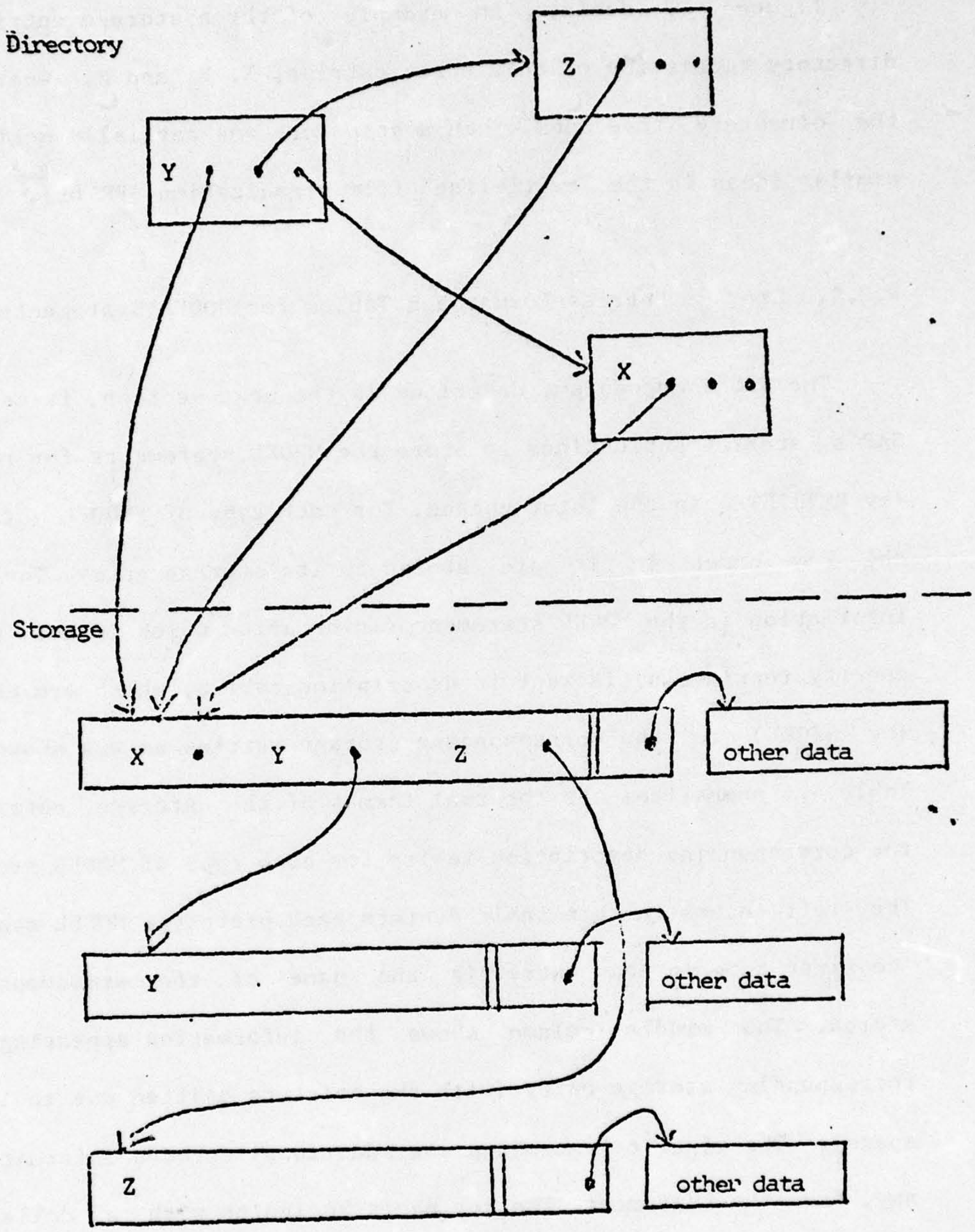


Figure 4.7

Sample Directory and Storage Entries

Table 4.4 Storage Entries Format for MODEL

MODEL Statement Schema	Storage Entry Key Names	Auxilliary Descriptions		
		Type	Stmt#.	
MODULE: module-name	module-name $MODULE	MODL	n	
SOURCE FILES: s_1, s_2,...,s_n	$SOURCE s_1 s_2 ... s_n	SRCF	n	
TARGET FILES: t_1, t_2, ... t_m	$TARGET t_1 t_2 ... t_m	TARF	n	
filename IS FILE(RECORD IS r, STORAGE IS s, KEY IS k)	filename r s k $FILE	FILE	n	key-flag (0=no sort key 1=sort key)
record-name IS RECORD (m_1, m_2,...,m_n)	record-name m_1 m_2 ... m_n $Pfile $RECD	RECD	n	#members members #subscripts first sub. second sub.
group-name IS GROUP (m_1,m_2,...,m_n)	group-name m_1 m_2 ... m_n $Pfile $GRP	GRP	n	(same as record)
report IS REPORT (REPORT_ENTRY IS r, STORAGE IS s, KEY IS k)	report r s k $REPT	REPT	n	(same as file)
report-entry IS REPORT_ENTRY (m_1, m_2, ..., m_n)	report-entry m_1 m_2 ... m_n $Pfile $RPTN	RPTN	n	(same as record)
field IS FIELD (fieldtype (minlength : maxlength))	fieldname $Pfile $FLD	FLD	n	fieldtype length min/max type 0=char 0=fixed n/m 1=binary 1=variable 2=numeric 3=decimal
interim IS INTERIM (same as for field)	interim $INTR	INTR	n	(same as for field)

MODEL Statement Schema	Storage Entry Key Names	Auxiliary Descriptions
assertion-name:	assertion-name s_1 $s_2 ... s_n$ $ASSERT	ASSR n #names components
SOURCE: $s_1, s_2, ..., s_n$;	assertioh-name t_1 $t_2 ... t_m$ $ASSERT	ASTG n #names components
TARGET: $t_1, t_2, ..., t_m$;	assertion-name $ASSERT	ASTX n assertion-text
storage-name IS		
CARD	storage-name $CARD	CARD n
TAPE (...)	$TAPE	TAPE n tape-attributes
DISK (...)	$DISK	DISK n disk-attributes
TERM (...)	$TERM	TERM n term-attributes
PUNCH(...)	$PNCH	PNCH n punch-attributes
PRINTER(...)	$PRNT	PRNT n print-attributes

Table 4.4 (Continued) Storage Entries Format for MODEL

4.2.4.4 The STORE Procedure

The STORE(S,D) Procedure has two parameters, S and D. S is the string containing the key names which are to be stored and to be entered in the directory. D is a pointer to previously built auxiliary data from the source string. The latter usually is an encoded form of non-key source language information.

Algorithm STORE shows the storing procedure. Section 4.2.3.2 already depicted the data structures that STORE creates.

STORE receives the key names from S and creates a storage entry for it (Steps 1-3). It checks if they are in the directory (Steps 4-5, subroutine SEARCH_DIR). If the key is in the directory, then it follows the "pointer-to-first" which points to the first storage entry with that name (Steps 7-8). The array of strings in each storage entry is scanned until the key name is found. If its "next" pointer is null (end-of-list), then it is set to point to the newly created storage entry (Steps 8-11). If it is not, the process is repeated until a null (end-of-list) pointer is found (Steps 9-10). If the current key name is not found in the directory, it is entered in the appropriate spot in the lexicographical position in the directory (Step 6, sub-routine CREATE_DIR) and the pointer in the directory is set to point to the

Algorithm STORE : The Store Procedure

Parameters: S=string of keys to be stored;
D=pointer to other data

(see Section 4.2.3.2 for diagrams of Data Structures)

[Subroutines called: CHECK_DIR, GENERATE_ENTRY]

```
Step 1. Count #KEYS.
Step 2. Allocate the storage entry for S (call it SE, according to the format shown).
Step 3. Connect PTR_TO_DATA in SE to D.
Step 4. For each key name, perform steps 5 through 11.
Step 5. If key exists in the directory (Algorithm CHECK-DIR ), then go to step 7; else go to step 6.
Step 6. Create a directory entry for this key. (Algorithm GENERATE-ENTRY )
Step 7. Let DE=this directory entry.
Step 8. If PTR_TO_FIRST in DE already points to a first storage entry with this key name, then go to step 9; else go to step 11.
Step 9. Get the next storage entry in the list.
Step 10. If it is the last in list, then go to step 11; else go to step 9.
Step 11. Add the new SE to the list.
Step 12. Return.
```

newly created first storage entry (Steps 7-8).

4.2.4.5 The RETRIEVE Procedure

RETRIEVE(E,D,S,N,P) is the procedure for retrieving desired storage entries, by searching through the data structures depicted in Figure 4.7 and Table 4.4. It is invoked by many routines described in subsequent phases of the Processor. It has five input parameters as indicated. RETRIEVE finds all the storage entries in which the given key name or expression of key names, E, appears and furthermore checks whether the first characters of data associated with the storage entries match the string D. That is, RETRIEVE finds all the storage entries with keys satisfying the logical expression E and other data D. RETRIEVE starts its search at directory entry S, normally the root node of the directory, and it returns a list of pointers P, to those storage entries which satisfy the request by the calling program. The number of storage entries satisfying the request is returned in N.

The logical expression E used to retrieve strings can be any boolean expression involving "key" names or names in the MODEL statements in disjunctive normal form, where the first key in each term is non-negated. For example, consider the following statement by a calling program:

```
CALL RETRIEVE(KEYS, '',START, N,P);
```

KEYS might contain the string value 'PRICE & ¬QUANTITY|EXTENT' . This makes RETRIEVE find all storage entries (which correspond to all statements in the MODEL specification) in which PRICE appears and

QUANTITY does not appear, or statements in which EXTENT appears. The null second parameter means that the auxiliary data portion of each statement is immaterial. RETRIEVE would then start its search and return a list of pointers in P to to those storage entries which satisfy the condition, and N would be set to the number of such statements that satisfy the condition.

Algorithm RETRIEVE is shown on the following page. An example showing the retrieval mechanism to retrieve all storage entries with key names "B" and "C" is given in Figure 4.7a. The diagram shows in parentheses the steps that correspond in the algorithm. RETRIEVE starts by getting the leading key name of the first conjunct (Step 1) and searches the directory for it (Step 2). If found, it puts the list of pointers to all storage entries with that name in a temporary list (Steps 3-7). If there are other names in the conjunct (Steps 10,14), then RETRIEVE eliminates the pointers in the temporary list to storage entries that do not have the other terms in the conjunct (Steps 14-16). If there are more conjuncts in the expression, then the process is repeated and the additional pointers are added to the list (Steps 12-13). When the end of the expression is reached, the list of pointers to the satisfying storage entries and the number of pointers are returned (Steps 20-22).

Algorithm RETRIEVE : The Retrieve Procedure

> Parameters: E=logical expression string; S=pointer to beginning of directory (input);
> P=list of pointers satisfying E; N=number of satisfying entries
>
> (see Section 4.2.3.2 for diagrams of data structures)

Step 1. Get leading key name K of next conjunct from E. If no more, go to Step 22.

Step 2. Check directory for K (standard binary tree search in subroutine SEARCH-DIP given earlier).

Step 3. If found, then go to step 4; else go to step 1.

Step 4. Set PSE=PTR_TO_FIRST (pointer to first storage entry with K)

Step 5. Add PSE to W list (temporary list of pointers)

Step 6. If K in PSE storage entry points to another storage entry with K, then go to step 7; else go to step 8.

Step 7. Set PSE to next storage entry in the list, go to Step 5.

Step 8. If end of E, then go to step 20; else go to step 9.

Step 9. Get next symbol in E.

Step 10. If symbol='&' then go to step 14; else go to step 11.

Step 11. If symbol='|' then go to step 12; else error return.

Step 12. Add list of pointers in W to list of pointers in P without duplication.

Step 13. Go to step 1.

Step 14. Get next symbol.

Step 15. If symbol='~' then go to step 16; else go to step 18.

Step 16. (Case of conjoining negated term) eliminate pointers in W to storage entries which also contain next key name in E.

Step 17. Go to step 8.

Step 18. (Case of conjoining non-negated term) eliminate pointers in W to storage entries which do not contain next key name in E.

Step 19. Go to step 8.

Step 20. Add list of pointers in W to list of pointers in P.

Step 21. Set N=number of pointers in P list.

Step 22. Return.

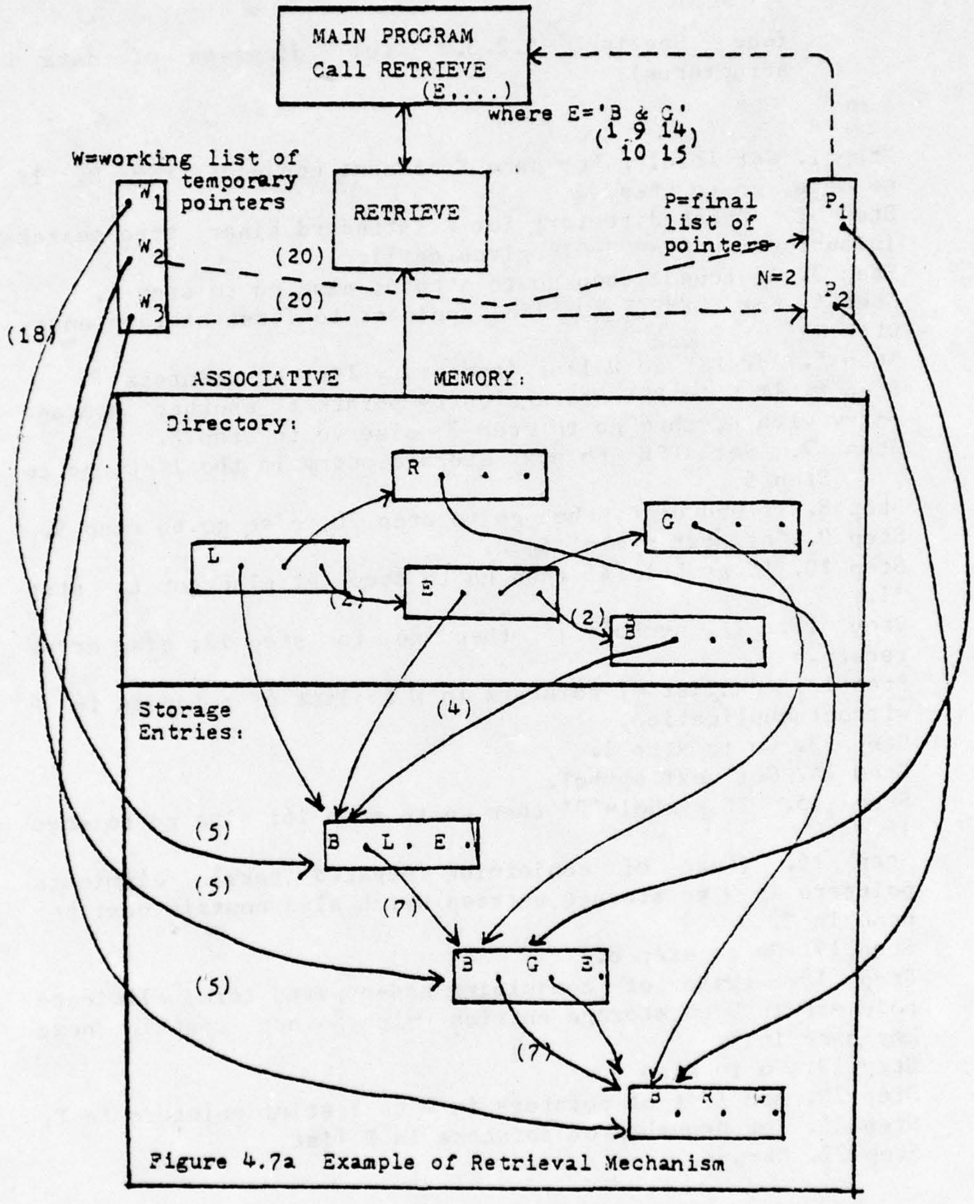


Figure 4.7a Example of Retrieval Mechanism

4.2.5 Components of a General System for Statement Analysis, Storage, and Retrieval

The SAPC, the lexical analyzer, and the storage and retrieval procedures form a self-contained sub-system that has been implemented and applied here to several tasks: analysis of MODEL statements for syntactic and local semantic correctness, storage of MODEL statements in a simulated associative memory, and a facility for later retrieval of stored statements for subsequent phases of the Processor. However, this collection of procedures in itself forms a general-purpose sub-system which can be used for processing languages other than MODEL. Nothing in this sub-system really depended on the nature of the MODEL language. With the exception of the lexical tables which might need adjustment for another language, the lexical analyzer, SAPG, STORE, and RETRIEVE could be applied directly as a general package for first pass processing of language statements.

4.2.6 Cross Reference and Attribute Report

A useful product of the Syntax and Statement Analysis Phase is a cross-reference report, produced by a cross-reference program (XREF) whose input is the encoded and stored MODEL specification. The XREF report provides an alphabetical listing of all the names provided by the user, and some of the reserved special names (such as CHOICE). For each name, the report provides the statement number in which the entity was described, the statement numbers of statements in which it is referenced, and the attributes or other known characteristics regarding the name.

For example, if field X is described in a given statement and is used in various other MODEL statements, such as in assertions, the cross-reference list would provide the original statement number in which it is described, a list of all the field's attributes as well as the names of the file or files in which it is a member, and a list of statement numbers which reference the given field name.

An example of a typical cross-reference report appears in Figure 4.8.

The cross-reference report is produced by the XREF module. It produces the report by traversing the directory and producing each line by successive uses of RETRIEVE to get the corresponding references. A bubble-sort is used to alphabetize the listing (in a subroutine named ALPHDIR).

```
              CROSS REFERENCE AND ATTRIBUTE REPORT

NAME        DESCRIPTION  ATTRIBUTES              REFERENCES
            STATEMENT

BALANCE     13           FIELD,CHARACTER,        7,41,43
                         FIXED,IN FILE
                         CUST
CHOICE                   RESERVED WORD           21,29,31,32,38,
                                                 22,25

CUST        2            FILE,SOURCE,            6,8,9,10
                         SORTED

CUSTDISK    7            DISK NAME

G           18           GROUP,4,IN FILE X       22,25

BALDUE      34           ASSERTION

NAME        15           FIELD,CHARACTER,        39,49
                         IN FILE X
NAME        20           FIELD,CHARACTER,        51,46
                         FILE Y

Z                        UNDEFINED-ERROR         52,59
```

FIGURE 4.8

EXAMPLE OF CROSS-REFERENCE AND ATTRIBUTE REPORT

4.3 Analysis of MODEL Specification

4.3.1 Introduction and Background

This phase of the MODEL Processor deals with the analysis of MODEL specifications by the use of graphs. It describes an application of graph theory to the analysis of MODEL specifications and to the generation of sequenced code from it. While graph theory has been used in various computer applications in recent years (e.g. [NUN71,LAN65]), the analysis of information relationships and automatic sequencing and generation of code by means of graphs have been novel and successful techniques here. This introductory sub-section presents and exemplifies the background and terminology involved in this phase, and describes the graphs, matrices, and other data structures that are built from a MODEL specification.

Section 4.3.2 provides an overview of the processes involved in this phase, and Section 4.3.3 discusses them in greater detail. In order to exemplify the algorithms and data structures used, a sample problem is presented below and described using the MODEL language. Its processing is traced throughout each of the sub-phases. The sample problem used is a small subset of the department store sale problem (DEPSALE) of Chapter 3. While the problem given in Chapter 3 presents a more realistic real-world situation, the smaller subset chosen is traced here because it is not overly complex, yet has enough features to exemplify the results of the algorithms.

As has been shown in the previous chapter, the statements of a MODEL specification consist of a series of descriptions of the following:

(1) files, each of which is designated as a source file, target file, or both;

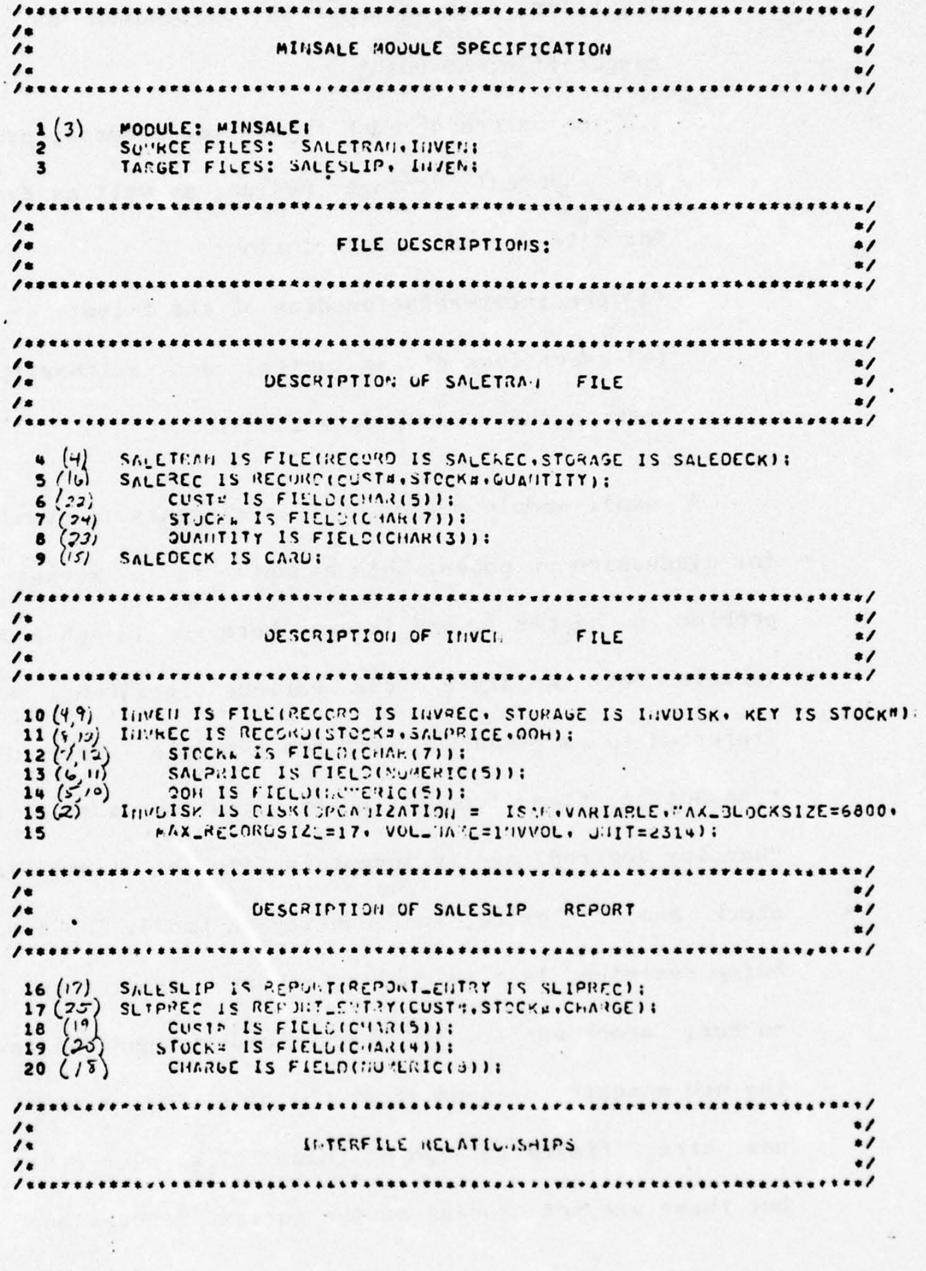
(2) components of each file; i.e. records, groups, fields and the physical storage medium, as well as dynamic assertions for data-dependent description;

(3) the inter-relationships of the files;

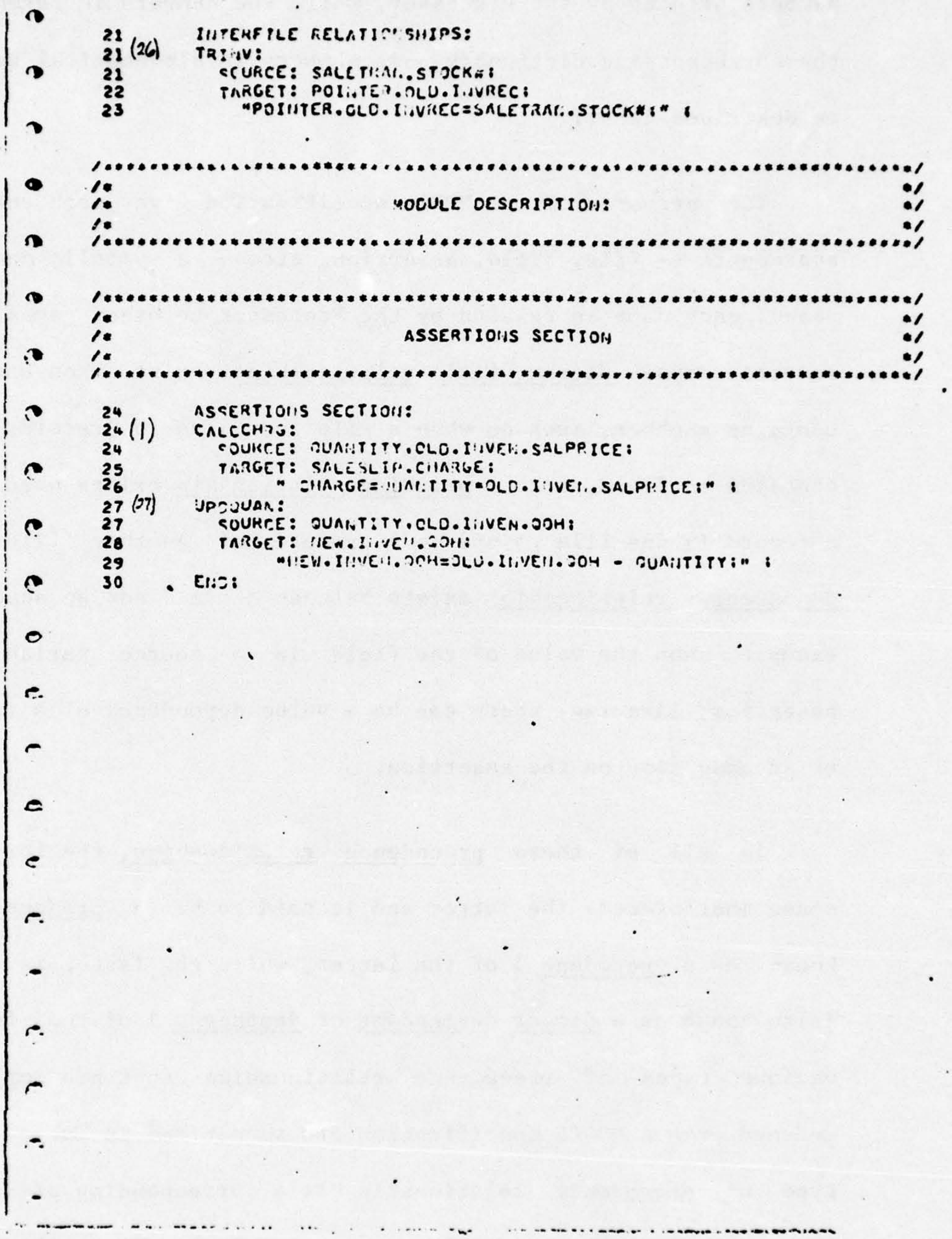
(4) assertions giving logical and arithmetic relationships between the various data items.

A small sample set of MODEL statements is provided in Figure 4.9 for discussion purposes. This example is a subset of the DEPSALE problem in Chapter 3, and is used here and in subsequent sections as a vehicle for explaining the various algorithms. The smaller example (referred to as MINSALE) describes a module whose input is a sale transaction file (consisting of a customer number, stock number, and quantity desired) and an inventory file of items (consisting of a stock number, price, and quantity on hand). The output of the module being described is a sale slip report (consisting of the customer number, stock number, and charge) and the updated inventory file with the new quantity on hand after the sale. The original DEPSALE problem has other fields in each of these files, plus other auxiliary files, but these are not crucial to the current discussion.

Figure 4.9 Sample Set of MODEL Statements



```
/*********************************************************************/
/*                                                                   */
/*                   MINSALE MODULE SPECIFICATION                    */
/*                                                                   */
/*********************************************************************/

 1 (3)   MODULE: MINSALE;
 2       SOURCE FILES:  SALETRAN,INVEN;
 3       TARGET FILES: SALESLIP, INVEN;

/*********************************************************************/
/*                                                                   */
/*                      FILE DESCRIPTIONS:                           */
/*                                                                   */
/*********************************************************************/


/*********************************************************************/
/*                                                                   */
/*                 DESCRIPTION OF SALETRAN   FILE                    */
/*                                                                   */
/*********************************************************************/

 4 (4)   SALETRAN IS FILE(RECORD IS SALEREC,STORAGE IS SALEDECK);
 5 (16)  SALEREC IS RECORD(CUST#,STOCK#,QUANTITY);
 6 (22)      CUST# IS FIELD(CHAR(5));
 7 (24)      STOCK# IS FIELD(CHAR(7));
 8 (23)      QUANTITY IS FIELD(CHAR(3));
 9 (15)  SALEDECK IS CARD;

/*********************************************************************/
/*                                                                   */
/*                 DESCRIPTION OF INVEN      FILE                    */
/*                                                                   */
/*********************************************************************/

10 (4,9)  INVEN IS FILE(RECORD IS INVREC, STORAGE IS INVDISK, KEY IS STOCK#);
11 (8,13) INVREC IS RECORD(STOCK#,SALPRICE,QOH);
12 (7,12)     STOCK# IS FIELD(CHAR(7));
13 (6,11)     SALPRICE IS FIELD(NUMERIC(5));
14 (5,10)     QOH IS FIELD(NUMERIC(5));
15 (2)    INVDISK IS DISK(ORGANIZATION =  ISAM,VARIABLE,MAX_BLOCKSIZE=6800,
15           MAX_RECORDSIZE=17, VOL_NAME=INVVOL, UNIT=2314);

/*********************************************************************/
/*                                                                   */
/*                DESCRIPTION OF SALESLIP   REPORT                   */
/*                                                                   */
/*********************************************************************/

16 (17)  SALESLIP IS REPORT(REPORT_ENTRY IS SLIPREC);
17 (25)  SLIPREC IS REPORT_ENTRY(CUST#,STOCK#,CHARGE);
18 (19)      CUST# IS FIELD(CHAR(5));
19 (20)     STOCK# IS FIELD(CHAR(4));
20 (18)      CHARGE IS FIELD(NUMERIC(8));

/*********************************************************************/
/*                                                                   */
/*                    INTERFILE RELATIONSHIPS                        */
/*                                                                   */
/*********************************************************************/
```



```
21        INTERFILE RELATIONSHIPS:
21 (26)   TRINV:
21          SOURCE: SALETRAN.STOCK#;
22          TARGET: POINTER.OLD.INVREC;
23            "POINTER.OLD.INVREC=SALETRAN.STOCK#;" ;


/***********************************************************************/
/*                                                                     */
/*                      MODULE DESCRIPTION:                            */
/*                                                                     */
/***********************************************************************/


/***********************************************************************/
/*                                                                     */
/*                           ASSERTIONS SECTION                        */
/*                                                                     */
/***********************************************************************/

24        ASSERTIONS SECTION:
24 (1)    CALCCHRG:
24          SOURCE: QUANTITY,OLD.INVEN.SALPRICE;
25          TARGET: SALESLIP.CHARGE;
26               " CHARGE=QUANTITY*OLD.INVEN.SALPRICE;" ;
27 (27)   UPDQUAN:
27          SOURCE: QUANTITY,OLD.INVEN.QOH;
28          TARGET: NEW.INVEN.QOH;
29               "NEW.INVEN.QOH=OLD.INVEN.QOH - QUANTITY;" ;
30        END;
```

The statement numbers down the left-hand side are MODEL statement numbers printed by the Processor, while the numbers in parentheses are the corresponding dictionary node numbers in alphabetical sequence to be described later.

The preparer of the MODEL specification gives each entity in his statements -- file, field, assertion, etc. -- a symbolic name. In this phase, each name is related by the Processor to other names in one of several ways. Hierarchical relationships exist when one data item contains another, such as when a file contains a record, a record contains a field, etc. A pointing relationship exists when a field of a record in one file points to a record of another file. A value dependency relationship exists between a field and an assertion, for example, when the value of the field is a source variable of the assertion; likewise, there can be a value dependency of a target field of an assertion on the assertion.

In all of these precedence relationships, the former in some sense must precede the latter and is said to be a predecessor (also known as a precedent) of the latter, while the latter is a successor (also known as a direct descendant or dependent) of the former. The various types of precedence relationships that are implicit in or deduced from a MODEL specification are summarized in Table 4.5. Each type of precedence relationship has a corresponding predecessor and successor type. The types of precedence relationships will have direct implications on the program to be generated. For example, a record must be read before any of its component fields can be used. A field

Precedence Number & Priority		Precedence Type	Predecessor Statement	Sucessor Statement	Explanation of Precedence
1	1	Hierarchical (source)	File, Record, or Group statement containing X as a member (in a source file)	Record, Group or Field Statement X (in a source file)	Predecessor contains sucessor as a data component (source file).
2	1	Hierarchical (target)	Field, Record, or Group statement X (in a target file)	File, Record, or Group statement containing X as a member (in a source file)	Predecessor is contained in sucessor as a data component
3	1	Explicit dependency	(a) Data description statement X or (b) Assertion whose target is X	(a) Assertion whose source is X or (b) Data description statement X	Predecessor is explicitly the source of successor assertion; or assertion has explicit target to successor data
4	2	Implicit dependency	First field statement identical to successor statement by name except in different files by the following rules: (a) Field in OLD source file (b) Field in source file (c) Interim (d) Field in target file	Field statement with no explicit predecessor	Predecessor is implicitly the source of successor due to lack of any other source and has the same name.
5	1	Pointing relationship	"Pointer type" assertion whose target is a record X being pointed to.	Record statement X	Predecessor serves as a symbolic pointer to record of a keyed file.
6	1	Media relationship	File statement whose medium is X	Storage medium description of X (DISK, TAPE, CARD, etc. statement)	Predecessor file stored on successor device.
7	1	Conditional dependence	Assertion with Target X which uses a function associated with a conditional flag	Statement X	Predecessor is explicit source of X but target complete only under a condition

TABLE 4.5 PRECEDENCE TYPES

pointing to a keyed record must be available before the record being pointed to can be accessed. A field which is a source or input to an assertion must be available and attain a value before invoking the procedure embodying the assertion. A field which is a target or output of an assertion is only available after the procedure is called. These and other requirements of the program to be generated are implied by the precedence information conveyed in a "directed graph" or "weighted adjacency matrix" as described below.

The entire aggregate of precedence relationships in a MODEL specification can be represented pictorially by a directed graph.

Formally, a directed graph is a pair <N,A>: a set of nodes N={N1,N2,...,Nm} and a relation A, i.e. a set of ordered pairs ("arrows" or "arcs") {A1,A2,...,Ap} where each Ai is an ordered pair (Nj,Nk) representing an arrow from node Nj to node Nk. In other words, A is a relation on N x N. Each node may have 0, 1, or more arrows emanating from it.

A weighted directed graph is a directed graph where each arrow (Nj,Nk) from node j to node k is a member of one of a set of different types of relations {R1,R2,...,Rq}.

Weighted directed graphs are used to represent precedence relationships in MODEL statements because of the different types of relationships, as described earlier. An example of a weighted directed graph appears in Figure 4.10, which corresponds to the example of Figure 4.9. Each node of this graph represents the name of one of the

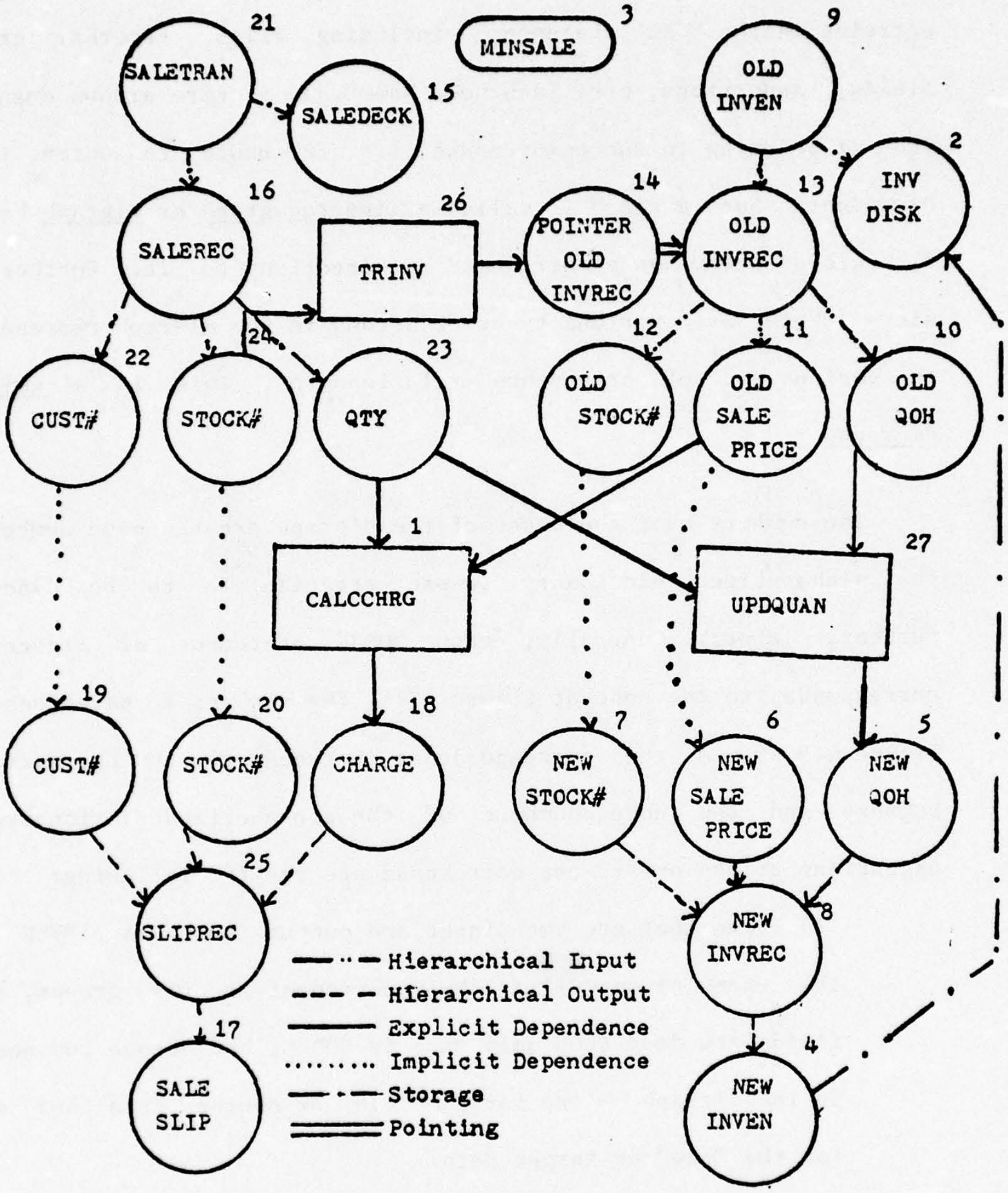


Figure 4.10
Digraph for MODEL Specification of Figure 4.9

entities in the MODEL statement, including files, records, groups, fields, assertions, etc. Each node has 0, 1, or more arrows emanating from it pointing to successor nodes; i.e. to nodes to which it is precedent. Such a graph is called a directed graph or digraph because each arc or arrow has an arrowhead or direction to it. Furthermore, since there are various types of arrows in the digraph representing the various type of precedence relationships, this is a weighted digraph.

The numbers over the nodes of the digraph are the node numbers of the alphabetized dictionary (whose creation is to be described further, later). Generally, each MODEL statement of Figure 4.9 corresponds to one node of Figure 4.10. The numbers in parentheses in Figure 4.9 showed the correspondence between the MODEL statement numbers and the node numbers of the alphabetized dictionary. The exceptions of the one-to-one correspondence are the following:

(1) Files that are both input and output (such as INVEN in the example) as well as their component records, groups, and fields are described only once in MODEL, but become two nodes in the digraph -- one for the "old" or source data and one for the "new" or target data.

(2) The list of source and target files in the header of the MODEL specification do not correspond to any node in the digraph because the file statements themselves correspond to the nodes for files.

(3) Special names in MODEL, such as POINTER names, EXIST names, etc., are not described explicitly by the MODEL user,

but are used only in context. However, in the digraph, such special names do correspond to nodes and have successors, predecessors, etc.

There are a number of non-pictorial representations of digraphs, which can be found in many standard textbooks [BER73]. One such formalization is called the adjacency matrix of a digraph. An adjacency matrix, A, corresponding to a digraph <N,R> of n nodes and one relation R, is an n x n matrix defined as follows:

Aij = 1 if (Nj,Nk) is in R; 0 otherwise

In order to distinguish between the different types of relationships that may exist between two nodes of the digraph, however, a weighted adjacency matrix is used. It has a zero everywhere that the adjacency matrix does, but has a number from 1 to 7 giving the type of relationship instead of the adjacency matrix entry of "1". The distinction between the different types of arcs in the digraph by use of these codes from 1 to 7 is used in later phases of analysis of this matrix. Formally, the weighted adjacency matrix is defined as follows:

Mij=k if (Nj,Nk) is in relation Rk; 0 if in no relation

The weighted adjacency matrix for the MODEL example of Figure 4.9 and for the corresponding digraph of Figure 4.10 is given in Figure 4.11. The node names are alphabetized and numbered down the left-hand side. The numbers to the right of the name are the original MODEL statement numbers from which these nodes are taken, as explained in the previous section. Entries in this matrix are either "0", indicating that there is no relationship between the row and column of the intersection, or are a code from 1 to 7 representing the type of relationship. The relationship codes of the weighted adjacency for MODEL are exemplified in Table 4.6. These codes correspond to the relationship types already discussed, and the examples are from the same sample problem. If Mij=k, the code in the left hand side of Table 4.6, then Ni is related to Nj in the manner shown in the middle column, with the examples appearing in the third column.

By showing predecessor and successor relationships, such a weighted adjacency matrix conveys all the precedence information of a MODEL specification. Note that in Table 4.6 hierarchical relationships "1" and "2" are reversed in direction because, for example, a record of an input file must be read before its component groups and fields are available, while the record of an output file must be written after its component groups and fields attain a value. For the same reason, the arrows emanating from nodes representing fields in output files were opposite in direction of those of input files in the pictorial digraph.

```
ADJACENCY MATRIX OF NAME RELATIONSHIPS

                                              5         10        15        20        25

 1  CALCCHRG                   27      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0
 2  INVDISK                    15      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 3  MINSALE                     1      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 4  NEW.INVEN                  10      0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 5  NEW.INVEN.QOH              14      0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 6  NEW.INVEN.SALPRICE         13      0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 7  NEW.INVEN.STOCK#           12      0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 8  NEW.INVREC                 11      0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 9  OLD.INVEN                  10      0 6 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10  OLD.INVEN.QOH              14      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
11  OLD.INVEN.SALPRICE         13      3 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12  OLD.INVEN.STOCK#           12      0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13  OLD.INVREC                 11      0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
14  POINTER.OLD.INVREC                 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15  SALEDECK                    9      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16  SALEREC                    16      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0
17  SALESLIP                           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
18  SALESLIP.CHARGE            20      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
19  SALESLIP.CUST#             18      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
20  SALESLIP.STOCK#            19      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
21  SALETRAN                    4      0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 1 0 0 0 0 0 0 0 0 0 0 0
22  SALETRAN.CUST#              6      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0
23  SALETRAN.QUANTITY           8      3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
24  SALETRAN.STOCK#             7      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 3 0
25  SLIPREC                    17      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0
26  TRINV                      21      0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0
27  UPDQUAN                    27      0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A NON-ZERO ENTRY IN ROW I & COLUMN J INDICATES THAT ITEM I PRECEDES ITEM J. THE CODE REPRESENTS THE FOLI

1 = HIERARCHICAL(SOURCE);  2 = HIERARCHICAL(TARGET);  3 = EXPLICIT DEPENDENCY;  4 = IMPLICIT DEPENDENCY:
5 = POINTING RELATIONSHIP;  6 = STORAGE RELATIONSHIP; 7 = CONDITIONAL DEPENDENCY
```

Figure 4.11 Weighted Adjacency Matrix for Sample MODEL Specification

Precedence Number	RELATIONSHIP TYPE	EXAMPLE
1	Hierarchical source node i is an input file, record, or group of an input file and contains node j as a sub-component	row of SALEREC and column of SALETRAN.CUST# has a 1 because the former contains the latter
2	Hierarchical target node i is a record, group or field of an output file and is contained in node j (its parent group, record, or file)	row of SALESLIP.CHARGE and column of SLIPREC has a 2 because the former is contained in the latter
3	Explicit Dependency node i is an explicit source of node j by virtue of a user assertion (latter is target of former)	row of QUANTITY and column of CALCCHRG has a 2 because the former is a source of the assertion
4	Implicit Dependency node i is an implicit source of node j due to implicit factors explained later	row of OLD.STOCK# and column of NEW.STOCK# has a 4 because there will be an implicit rule to this effect
5	Pointing Relationship node i is a pointer to node j (a record name).	row of POINTER.INVREC and column of INVREC has a 5 because the former points to the latter
6	Media Relationships node i is a file name stored on a medium whose name is in node j	row of INVEN and column of INVDISK has a 6 because the former is stored on the latter
7	Conditional Relationship	None in this Example

Table 4.6 Illustration of Precedence Relationships of Table 4.5 for Matrix of Figure 4.11

All such precedence information is built into the matrix and analyzed in subsequent sections.

4.3.2 Overview of Sub-phases in Network Creation and Analysis

The digraph of a set of MODEL statements and its representation as a Weighted Adjacency Matrix is a crucial factor in the MODEL Processor's ability to sequence operations and to detect many inconsistencies and incomplete specifications that a user might submit. Table 4.7 shows a summary of the nine steps or sub-phases involved in the creation and analysis of the matrix representation of the digraph (or "network"). It summarizes the tasks of each step and the relationships for which each step searches. The nine sub-phases themselves are described in greater detail in sub-sections 4.3.3.1 through 4.3.3.9. The process is begun by first creating a dictionary of names that is to correspond to node numbers of the matrix (Sub-phase 1, Section 4.3.3.1). Creation of the matrix (Sub-phase 2, Section 4.3.3.2) and entering the various precedence relations within it are dealt with next (Sub-phases 3 through 7, Sections 4.3.3.3 through 4.3.3.7). The last two steps deal with more graph analysis (Section 4.3.3.8) and cycle detection (Section 4.3.3.9).

One of the major tasks during this entire phase is detecting logical errors and reporting them to the user. In parallel to searching and entering precedence relationships in the weighted adjacency matrix, certain kinds of logical errors are detected, and messages are sent to the user. Further error analysis takes place after the Processor constructs the matrix. A summary of the error conditions that are searched for by each of the nine sub-phases is summarized in Table 4.7a. This table also refers to the error messages

Step Name	Summary of Tasks and/or Relationships Searched
1 Creating Dictionary (CRDICT)	Creates a dictionary of all names assigning a "node" number to each
2 Creating Weighted Adjacency Matrix (CRADJMT)	Creates n x n weighted adjacency matrix (initialized to zeros) whose rows and columns correspond to the dictionary
3 Entering Hierarchical Relationships (ENHRREL)	Searching for hierarchical relationships between a parent and descendant data
4 Entering Explicit Value Dependencies (ENEXDP)	Searches for explicit value dependency relationships given by assertions
5 Entering Conditional Value Dependencies (ENEXDP)	Searches for explicit value dependencies with a conditionally completed node
6 Finding Implicit Predecessors (FNDISRC)	Searches for implicit predecessor to nodes with no explicit predecessor
7 Entering Pointing Relationships (ENPTREL)	Enters pointing relationships between pointer names and records
8 Graph Analysis (AMANAL)	Analyzes the weighted adjacency matrix to ensure that certain error conditions do not exist (See Table 4.7a)
9 Cycle Detection (CRPATHS,CYCLES)	Creates path matrix & searches for possible cycles

Table 4.7

Steps in Network Creation and Analysis

Step Name	Condition Triggering Error	Msg# & Examples Tab.4.6c
1 Creating a Dictionary (CRDICT)		
2 Creating the Weighted Adjacency Matrix (CRADJMT)		
3 Entering Hierarchical Relationships (ENHRREL)	Multiple, contradictory descriptions of data name.	4
	missing description of a data descendant	6
4 Entering Explicit Value Dependencies (ENEXDP)	An assertion source is never itself described;	1
	An assertion target is never itself described	2
5 Entering Conditional Value Dependencies (ENEXDP)	(same as for 4)	
6 Finding Implicit Predecessors (FNDISRC)	A field in a target file or interim has no explicit predecessor & no deductions cited in FNDISRC could be made	3
	An implicit predecessor found & assumed assn. generated	10
	Several possible implicit predecessors found, but one chosen; assumed assn. generated	11
7 Entering Pointing Relationships (ENPTREL)	An assertion of the form POINTER.R=F given, but R is not any record	13
8 Graph Analysis (AMANAL)	2 or more source files are not properly related	5
	Field in source file unused	8
	Field in source file is the target of an assertion	12
	Several assertions have same target field (must be mutually exclusive conditions)	9
9 Cycle Detection (CRPATHS, CYCLES)	Circular statements exist	7

Table 4.7a
Error Conditions Detected during Network Creation and Analysis

that are produced in each case and to an example of the error condition by means of an error number.

Table 4.7b summarizes, by number, the messages themselves which can possibly be produced by the Processor during the nine sub-phases in the matrix creation and analysis phase and gives examples of situations that give rise to these messages.

Table 4.7b

Summary of Errors Detected During Entire Matrix Analysis Phase

Message 1:

ERROR(INCOMPLETENESS):
Need to know how to obtain X in assertion A.

When Issued/example:

If X is a source to A but never itself described.
Example:

```
A: SOURCE: X;
   TARGET: Y;
```

but description of X not given.

Issued by routine: ENEXDP (Step 9)

Message 2:

ERROR(INCOMPLETENESS):
Need to know how to use X in assertion A.

When Issued/Example:

If x is a target of assertion A but never itself described.
Example:

```
A: SOURCE: W;
   TARGET: X;
```

but description of X not given.

Issued by routine: ENEXDP (Step 9)

Message 3:

ERROR(INCOMPLETENESS):
Need to know how to obtain field X.

When Issued/Example:

If field X is in a target file or an interim, but no assertion exists that describes how X is obtained and nothing can be deduced.
Example:

```
X IS FIELD(...)
```

where X is a field in a target file, but no assertion exists which obtains X.

Issued by routine: FNDISRC (Step 5)

Message 4:

ERROR(INCONSISTENCY):
X is described more than once [Contradictory descriptions of X].

When Issued/Example:

(1)If X is described in 2 or more data description statements in the same file:
Example:
X IS FIELD(CHAR(2));
X IS FIELD(NUMERIC(9));
where both pertain to the same file; or

(2) If X is described as 2 or more files, or assertions, etc. at the same time:
Example:
X IS FILE(...);
X: SOURCE: ...; TARGET...;

Issued by routine: ENHRREL (Step 15)

Message 5:

ERROR(INCOMPLETENESS):
Files F1, F2,... Are not related.

When Issued/Example:

When Files F1, F2,... Are source files but are not in any way related (by POINTERS, etc.).

Issued by routine: AMANAL (Step a)

Message 6.

ERROR(INCOMPLETENESS):
Description of Group or Field X in Y missing.

When Issued/Example:

Y (a file, record, or group) is described to have descendant X, but X is nowhere described.
Example:
Y IS RECORD(X,V,U);
V IS FIELD(...)
U IS FIELD(...)
i.e. description of X is missing.

Issued by routine: ENHRREL (Step 14)

Message 7:

ERROR (INCONSISTENCY):
The following groups of items are circularly described:
...

When Issued/Example:

When items are described circularly.
Example:

```
A: X=Y+Z;
B: V=X+W;
C: Y=V+U;
```

Issued by routine: PRCYCLES (which is called by CYCLES enumeration, Step 24).

Message 8:

WARNING (POSSIBLE INCOMPLETENESS):
Nothing is obtained from X.

When Issued/Example:

X is a field in a source file or is an interim name, but it is never used elsewhere in the specification.
Example:

```
X IS FIELD(...);
```

X is never used elsewhere in this specification of the module (intentionally or inadvertently).

Issued by routine: AMANAL (Step b)

Message 9:

WARNING (POSSIBLE AMBIGUITY).
X is given a value by assertions A1, A2, ...; they must be under mutually exclusive conditions.

When Issued/Example:

More than one assertion describes how X is obtained; may be alright if under mutually exclusive conditions.

Example:

```
A1: SOURCE: CHOICE.C1,Y;
    TARGET: X;
    ...
A2: SOURCE: CHOICE.C2, W;
    TARGET: X;
    ...
```

This could be alright if C1 and C2 are mutually exclusive.

Issued by routine: AMANAL (Step c)

Message 10:

WARNING(APPARENT INCOMPLETENESS):
Following assertion assumed:
"X=Y"

When Issued/Example:

When
(1) X was not assigned a value by means of an explicit assertion; and
(2) it was possible for the Processor to find an implicit predecessor using the first applicable of the following rules:
(a) X is in a file which is both source and target, so OLD name is assigned to the NEW name.
Example: NEW.X=OLD.X;

(b) Y has the same name as X, except that Y appears in one of the source files.
Example: F.X=G.X
where F is the target file, and G is the source file with the same-named field.

(c) Y has the same name as X, and Y is an interim field.
Example: F.X=INTERIM.X;

(d) Y has the same name as X, and Y is in another target files and already has a value itself.
Example: F.X=G.X;
where G is another target file with the same-named field, which already has a value assigned to it.

Issued by routine: FNDISRC (Rules 1-4)

Message 11:

WARNING(APPARENT AMBIGUITY):
Following assertion is assumed:
"X=Y;"

When Issued/Example:

When
(1) X was not assigned a value by means of an explicit assertion; and
(2) the Processor determined an implicit predecessor using the first applicable of the following rules:

(just like the previous set of messages, except that here there is more than one candidate for a predecessor, because of multiple same-named fields in different files, so the first such candidate found is arbitrarily chosen and printed to the user).

(a) (see 10b).

(b) (see 10d).

Issued by routine: FNDISRC (Rules 1-4)

Message 12:

ERROR(INCONSISTENCY):
Field X is a source-file field and cannot be the target of assertion A.

When Issued/Example:

When X is described to be in a file that is source to the module and X is described to be the target of an assertion.
Example:

```
SOURCE FILES: F,...;
...
F IS FILE(...);
  X IS FIELD (...); (in file F)
A: SOURCE: Y;
   TARGET: X;
```

Issued by: AMANAL (Step d)

Message 13:

ERROR(INCONSISTENCY):
POINTER.R is used but R is not the name of any record.

When Issued/Example: When a POINTER type assertion of the form POINTER.R=F is given, but R is not the name of any record.

Issued by: ENPTREL (Step 4)

4.3.3 Sub-phases of Network Creation and Analysis

This section supplies greater detail on each of the sub-phases of network creation and analysis, and the logical errors that are detected by each phase. Peferences are made to the message numbers of Table 4.6c.

4.3.3.1 Creating a Dictionary for Row and Column Numbers of the Weighted Adjacency Matrix

The "Create Dictionary" (CRDICT) procedure creates a dictionary of names, assigning a "node" number to each. These names correspond to the nodes of the digraph and they become the rows and columns of the Weighted Adjacency Matrix. The dictionary data structure (DICT) is an array of strings. An entry is made in the dictionary for each distinct, fully qualified name of each file, record, group, field, interior, storage device, or assertion named in the user's MODEL specification, each name roughly corresponding to a statement in the specification. For example, a field name entry corresponds to a field description statement, an assertion name entry corresponds to an assertion statement, etc.

However, there are exceptions to the correspondence between dictionary names and statements in MODEL. If a file is described in MODEL to be both a source and target file, its component record, groups, and fields (described once in the MODEL specification) appear in two separate entries in the dictionary (DICT) because they represent two distinct entities ("OLD" and "NEW"). Furthermore, there are several types of "special names" in a MODEL specification that can be the source or target of an assertion (with certain restrictions as explained in Chapter 3), and which become entries in the dictionary. These include POINTER names, EXIST names, LEN names, CHOICE names, and SUBSET names (as described in Chapter 3). Such special names are never explicitly described in a data description statement as fields are, since their description is implicit. They do however become nodes in the digraph (rows and columns in the matrix) and therefore need dictionary entries. Assertions whose sources or targets are one of these special names are treated in a special way in code generation as shown later.

Algorithm CRDICT shows the details of the Create Dictionary Procedure. It goes through each entry of the directory and retrieves the corresponding statement (Steps 1-3). Each name is fully qualified with the filename, "OLD" or "NEW" qualifiers, etc. and is entered in the dictionary (Steps 4-9). It also creates entries for the special names explained above (Step 10). The dictionary is alphabetized at the end (Step 11) and each name then has a unique node number corresponding to it.

Algorithm CRDICT: Creating the Dictionary

[Subroutines called: RETRIEVE]

Step 1. Get next directory entry.

Step 2. If there are more directory entries, then go to Step 3; else go to Step 10.

Step 3. RETRIEVE statements (storage entries) in which the name is described.

Step 4. Branch on statement type:

RECD, then go to Step 8;
INTR, then go to Step 6;
FLD or GRP, then go to Step 7;
FILE, then go to Step 8;
Others, then go to Step 5.

Step 5. Enter name in next entry of dictionary as is; go to Step 1.

Step 6. Qualify Interim Name by prefixing it with "INTERIM." and enter in next entry in dictionary; go to Step 1.

Step 7. Qualify name with its parent file; go to Step 8.

Step 8. If corresponding file is both a source and a target file, then go to Step 9; else go to Step 5.

Step 9. Enter name in dictionary twice: once with "NEW." and once with "OLD." prefix; go to Step 1.

Step 10. Using RETRIEVE, find all CHOICE, EXIST, LEN, POINTER, & SUBSET names and enter each one in the dictionary once.

Step 11. Alphabetize the dictionary (using standard bubble sort).

Step 12. Return.

4.3.3.2 Creating the Weighted Adjacency Matrix and Entering Precedence Relationships Within It

The collection of user-provided names from the specification which now appears in the dictionary, forms the rows and columns of the weighted adjacency matrix. That is, the weighted adjacency matrix, M, is allocated as an n x n matrix whose rows and columns correspond to the n user names appearing in the MODEL specification.

Algorithm CRADJMT shows the steps of the procedure "Create Adjacency Matrix" (CRADJMT). It outlines the creation of the Weighted Adjacency Matrix, including its allocation (Step 1), its initialization (Step 2), and the invocation of subroutines that detect and enter precedence relationships within it (Steps 2-4). The matrix is initialized to all zeros indicating no relationship between node i and node j (for all i,j) as a default. The procedure then proceeds to call other routines which detect and enter precedence relationships of various types. In these subsequent procedures, values are entered in the rows and columns of the weighted adjacency matrix by analyzing relationships in the MODEL specification submitted by the user. Many of these relationships are explicit in the user statements, while others are implicit and deduced by the Processor. Furthermore, certain kinds of logical inconsistencies and incompleteness in the MODEL statements can be detected during the construction and analysis of matrix M.

Algorithm CRADJMT: Creating the Weighted Adjacency Matrix and Entering Precedence Relationships Within it

[Subroutines called: ENDPREL,ENHRREL,ENPTREL]

```
Step 1. Allocate the Weighted Adjacency Matrix, M as an nxn matrix.
Step 2. Set Mij=0 (for all i,j).
Step 3. Call ENDPREL (Enter Value Dependency Relationships).
Step 4. Call ENHRREL (Enter Hierarchical Relationships).
Step 5. Call ENPTREL (Enter Pointing Relationships).
Step 6. Return.
```

4.3.3.3 Entering Hierarchical Relationships

Hierarchical relationships are entered in Matrix M between files, records, groups, and fields by the routine named ENHRREL (ENter HieRarchical RELationships). Table 4.6 showed that if item i contains item j, both of which are items in an input file, then a "1" is entered in the matrix row of i and the column of j, whereas if item i is contained in item j, both of which are in an output file, then a "2" is entered in the row of i and the column of j. This is due to the fact that the precedence is opposite in direction for input and output files.

Algorithm ENHRREL shows the procedure to enter hierarchical relationships. Entering the hierarchical codes is accomplished by retrieving all the file descriptions (Steps 1-2) and successively finding the components of each. By means of a recursive procedure (Step 4, ENT_HIER_ADJ) that "climbs" down the implicit hierarchic data structure, each component's direct descendant statements are retrieved in turn and the hierarchical relationship between a parent and its direct descendants is successively entered in the matrix M (Steps 1-7 of ENT-MED-ADJ). Formally, if node i is the parent of node j (e.g. node i is a record containing a field node j) then

Mij = 1 if the current file is an input file

Mij = 2 if the current file is an output file

Furthermore, if node j is not a lowest level field (Step 10), then its descendants are found, in turn, and the procedure is invoked recursively to insert the hierarchical relationships with their

Algorithm ENHRREL: Entering Hierarchical Relationships

[Subroutines called: RETRIEVE,ENT_HIER_ADJ,ENT_MED_ADJ]

```
Step 1. RETRIEVE all files or reports.

Step 2. Get next file or report.

Step 3. If none, then go to Step 6,else go to Step 4.

Step 4. Call ENT_HIER_ADJ (Filename,Recname).
(a recursive procedure to climb down the data structure tree, starting
with file and record).

Step 5. Call ENT_MED_ADJ (Filename,Medium name)
(a routine to enter relationship between file and medium).

Step 6. Return.
```

Enter Hierarchical Relationships in Weighted Adjacency Matrix (a recursive routine)
[Subroutines called: RETRIEVE]

```
Step 1. Qualify parent and direct descendant names.

Step 2. Let i=dictionary number of parent.

Step 3. Let j=dictionary number of direct descendant.

Step 4. If current file is source only, then go to Step 5;
if current file is target only, then go to Step 6;
if current file is source and target, then go to Step 7.

Step 5. (source only) Set Mij=1 (hierarchical input code);go to Step 8.

Step 6. (target only) Set Mji=2 (hierarchical output code);go to step 8.

Step 7. (source and target)
Set i=dictionary number of "OLD" parent.
Set j= dictionary number of "OLD" direct descendant.
Set Mij=1.
Set i=dictionary number of "NEW" parent.
Set j=dictionary number of "NEW" direct descendant.
Set Mji=2.

Step 8. RETRIEVE direct descendant storage entry.

Step 9. If one direct descendant storage entry is found, then go to Step 10;
if no direct descendant storage entry found, then go to Step 14;
if more than 1 direct descendant storage entry found, go to Step 15.

Step 10. If type of direct descendant is record, group, or report entry, then go to Step 11;
if type of direct descendant is field, then go to Step 13; else system error.

Step 11. Get all of its direct descendants.

Step 12. For each one, call ENT_HIER_ADJ recursively to enter hierarchical relationships between it & its descendants (go to Step 1).

Step 13. (field: no further direct descendants) Return.

Step 14. Print incompleteness message (#6); go to Step 16.

Step 15. Print inconsistency message (#4); Go to Step 16.

Step 16. Return.
```

descendants into the matrix (Steps 11-13).

Note that hierarchical relationships "1" and "2" are reversed in direction for precedence purposes (Step 7) because, for example, a record of an input file must be read before its component groups and fields are available, while the record of an output file must be written after its component groups and fields attain a value.

Certain errors can be detected during this process (Steps 14-15). If at a given node the indicated descendants do not exist and therefore cannot be retrieved (e.g. if a record X is described to have fields A and B but field B is never described), then the file layout is poorly-defined due to incompleteness. Likewise, if at a given node more than one descendant with the identical name can be found in the same file (e.g. field X of a given file is described twice with two different sets of attributes), then the file is ill-defined due to an inconsistency. Such problems are reported to the user in the Network Analysis Report in a manner similar to the following (Message numbers 6 and 4, respectively):

ERROR(INCOMPLETENESS): Need a description of X

or

ERROR(INCONSISTENCY): X is described more than once.

After entering all the hierarchical relationships in matrix M, the storage media relationships are entered by the ENter MEDia ReLationships routine (Step 5; ENT-MED-ADJ) . If a file corresponding to node i is stored on a storage medium with name corresponding to node j, then Mij=6, a code indicating the storage relationship.

4.3.3.4 Entering Value Dependency Relationships

Value dependency relationships are entered in M to indicate that an item, j, such as a field or assertion depends on the value of another item, i, and that therefore item i is precedent to item j. These relationships are detected and entered by the routine ENDPREL (ENter DePendency RELationships). Some dependency relationships are explicit in the MODEL statements, while others are implicit and are deduced or assumed by the Processor.

Algorithm ENEXDP shows how value dependency relationships are detected and entered in the weighted adjacency matrix. For every explicit assertion, R, appearing in the MODEL statements (Steps 1-2), there are "source" fields on which the assertion R depends, and "target" fields to which assertion R is precedent. For every source field which corresponds to node i, an "explicit value dependency code" of 3 is entered in matrix M in row i and the column corresponding to assertion R in order to indicate that source field i is precedent to assertion R (Steps 5-8). Likewise, for every "target" field which corresponds to node j, the code of 3 is entered in the matrix in the row corresponding to assertion R and column j, in order to indicate

Algorithm ENEXDP: Detecting and Entering Explicit Value Dependency Relationships

[Subroutines called: RETRIEVE]

```
Step 1. Retrieve all assertions.
Step 2. For each assertion perform Steps 3 through 20.

    Step 3. For each source and target to the assertion, perform
    Steps 4 through 19.
        Step 4. If assertion uses special REPLACE function, then
        go to Step 10;
        if assertion uses a conditional function, then set code=7
        (conditional value dependency code);
        else set code=3 (normal explicit value dependency code).

        Step 5. (normal case: Steps 5 through 8):
        Set k=dict. no. of source or target name.
        Step 6. If found then go to Step 7; else go to Step 9.
        Step 7. Let l = dictionary number of assertion.
        Step 8. If name is source to assertion, then set
        A(k,l)=code;
        if name is target, set A(l,k)=code;
        go to Step 14.
        Step 9. Print incompleteness error (Message #1 if source,
        #2 if target); go to Step 14.

        Step 10. (Replace function case; Steps 10-13):
        Step 11. Set k=dict-no of EMPTY condition.
        Step 12. Set l=dict-no of current assertion.
        Step 13. Set A(l,k)=code. (to insure REPLACE execution
        precedes EMPTY evaluation).
        Step 14. If assertion is specifying a SUBSET of a target
        file then go to step 15; else go to step 19.

        Step 15. (subset case; steps 15-18):
        Step 16. Set l = dictionary number of this assertion.
        Step 17. Set k = dictionary number of record corresponding
        to file whose subset is specified.
        Step 18. Set A(l,k)=code (to ensure that SUBSET condition
        evaluated before the WRITE command).

        Step 19. Try next source or target name (go to step 4).

    Step 20. Try next assertion (go to step 3).

Step 21. Return.
```

that assertion R is precedent to target field j (Steps 5-8).

Note that this algorithm also handles various special cases, such as assertions that use conditional functions (Steps 4-8) described in the next section.

Also note that other than a field or interim, a source or target of an assertion can also be a "special name" not explicitly described by the specifier, such as CHOICE names, EXIST names, LEN names, etc. explained in Chapter 3. Since such names correspond to rows and columns in the weighted adjacency matrix, their precedence is still entered in the same way as field sources and targets.

During this ENEXDP process of entering the explicit dependencies in M, certain kinds of user incompleteness in the specification can be detected (Steps 6,9). For example, if a field "X" is the source or target of some assertion, but is never itself described as a field of any file or interim field, then the assertion is ill-defined because it refers to a non-existent field and the specification is incomplete as field X is never described. A message is sent to the user in the Network Analysis Report to the effect that there is an "Incompleteness: Need to know how to use (or obtain) X", and the user is directed to provide the missing description or change the statement with the faulty reference, whichever is appropriate (Message numbers 1 and 2). Similar errors are detected when a target of an assertion is itself undefined.

4.3.3.5 Entering Conditional Dependency Relationships

The above routine, as shown in Algorithm ENEXDP, also has the task of entering conditional relationships in the adjacency matrix.

An assertion written by the user can utilize any of the system-provided functions. The completion of some of these functions, however, is dependent on an associated condition. An example of such a system-provided function that is "conditionally completed" is a summation function where quantities are summed over different cross-sections of a file. The output of that function, however, is not considered complete until an associated condition is met; in this case, that all records in the range of the summation are processed. The system function itself is responsible for setting a flag at execution-time when the function is considered to have been completed. A list of all such system-provided functions appears in the Processor in a system table called SYSFCN, a table containing the names of the various system functions.

The ENEXDP (ENter EXplicit DePendencies) algorithm enters the "conditional value dependency code" of 7 (rather than the explicit value dependency code of 3) from an assertion to its target, whenever the assertion uses a conditionally-completed function to generate the target (Algorithm ENEXDP Step 8).

4.3.3.6 Finding Implicit Predecessors

If a field that is not in some input file does not have a value via some explicit user's assertion, then the Processor next tries to find a implicit source for the field using a set of successive rules. Also during the following phase, further analysis is made of the Weighted Adjacency Matrix, M, and certain kinds of inconsistency and incompleteness errors are detected. Details of entering such implicit relationships in the adjacency matrix and detecting corresponding errors are in the process called ENtering IMplicit DePendence (ENIMDP), and its subroutines, described here.

First, interim variables are checked to make sure that they have a predecessor. The HASSRC ("HAS SouRCe") function determines whether an item has an explicit predecessor. If an interim field corresponds to node j, then column j of M is checked to see if it has an explicit predecessor; i.e. $(\exists i)(Mij>0)$. If so, then the field has a source; otherwise, a message such as the following is sent to indicate its absence (Message number 3):

> ERROR(INCOMPLETENESS): Need an assertion that describes how to obtain interim name X.

Secondly, all the fields in target files are checked to determine whether they already have an explicit predecessor via the HASSRC function. If a given field in a target file (a field corresponding to, say, node j in Matrix M) already has an explicit source by virtue of a user's assertion, then $(\exists i)(Mij=3)$. Otherwise, the field has no explicit source and the Processor tries in the FNDISRC routine (FiND

Implicit SouRCe) to find a same-named field in another file or a same-named interim field as its source using a set of successive rules in the following order of priority. Although there might be other possible and equally reasonable rules for predecessor assumptions, they could easily be incorporated by another systems programmer. The idea here is for the Processor to make some reasonable assumption for a plausible predecessor if at all possible. Regardless of the Processor assumptions, the user can modify the result of the assumption in his next specification iteration by removing, changing, or adding assertions. The following rules are used by the FNDISRC Algorithm.

Rule 1: If the target field having no explicit predecessor is in a file which is both a source and target file, then the value in the corresponding field in the old record is taken as the value of the field in the new record (Message 10 is printed).

Rule 2: If Rule 1 does not apply, then the Processor tries to find a same-named field in a source file. If one is found, it is assumed to be the source and is so indicated in a message containing the assumed assertion (Message 10). If more than one same-named field in a source file is found, then the first is taken as a source and a message is sent to indicate that there was an ambiguity, and the assumed assertion is printed (Message 11).

Rule 3: If no predecessor for the field is found by the above means, then the Processor tries to find a same-named interim field. If one is found, it is taken as the source and a message is sent to indicate that (Message 10). If more than one is found, the first is taken and a message is sent to indicate that there was an ambiguity (Message 11).

Rule 4: If the above efforts are unsuccessful, the Processor tries to find a same-named field in another output file. If one is found it is taken as the source with a corresponding message given to the user (Message 10), and if more than one is found, then one is taken with a corresponding message to the user regarding the ambiguity (Message 11).

Rule 5: In the above cases, the Processor tries to find "implicit" sources for a field if none is given explicitly. If all this still fails to find some field which can be construed to represent the current field's source, then an error message is sent to the user to the effect that the current field has no assertion describing how it is obtained, and that therefore such an assertion is needed (Message 3).

In the above cases where an assumption is made regarding an implicit precedence, the corresponding assertion is printed to the user in his own language, namely in the form of a MODEL assertion. A warning is printed as follows: "In the absence of any other relationship, the following assertions have been assumed:", followed by the assumed assertions. The warning (Messages 10 and 11) is produced by the PRSRCWRN routine (PRint SouRCe WaRNing).

The resulting list of such assumed assertions can then become a permanent part of the documentation by appending them to the listing. The assumed assertion is written out in the user's own MODEL language for him to evaluate whether it agrees with his own original intention or whether some of the statements must be changed and the specification resubmitted. If the user is satisfied with the assumed assertions and if he wishes that they be explicitly incorporated into the specification's original assertions, he could use an on-line editor to merge the assertions produced by the Processor with his own.

4.3.3.7 Entering Pointing Relationships

This phase of the Processor has the simpler task of entering "pointing" relationships in the Matrix M. Such a relationship exists when a user states that some field points to some record of a file. This is stated in MODEL with an assertion of the form "POINTER.R=F," where R is the name of some record and F is the name of the pointing field.

For example, in the subset of the Department Store Sale problem presented earlier in this chapter, there is a pointing relationship between POINTER.INVREC and INVREC. A "pointing code" of 5 is then entered in the matrix to show the precedence of the pointing field to the keyed record. If there is no such record name as "R", then an error message is sent to that effect to the user (Message 13).

Algorithm ENPTREL shows the procedure to enter all such pointing relationships into the matrix:

```
Algorithm ENPTREL: Entering Pointing Relationships:
[Subroutines called: RETRIEVE]
Step 1. Retrieve all POINTER names.
Step 2. For each POINTER name, perform steps 3 through 5.
Step 3. Set i=dictionary number of pointer name.
Step 4. Set j=dictionary number of record pointed to. If
missing, print error message (#13).
Step 5. Set Mij=5 (pointing code).
Step 6. Return.
```

4.3.3.8 Graph Analysis of Adjacency Matrix

After entering all the known precedence relationships into the weighted adjacency matrix, the matrix such as the one shown in Figure 4.10 is printed out for the user by the PRADJMT routine (PRint ADJacency MaTrix). The dictionary of names appears in alphabetical order (having been sorted by ALPHDIR) for the user alongside the corresponding rows of the matrix.

Although by this time many logical errors in the MODEL statements have been detected during the construction of M, such as the inconsistencies, ambiguities, and incompleteness explained in the previous sections, some of the analysis can be done only after the construction of matrix M is complete.

Some examples of the analysis performed at this stage are as follows:

(a) If a given row, i, of matrix M corresponds to a field that has no direct descendants, i.e.

$(\nexists j)(M_{ij}=3 \text{ or } M_{ij}=4)$

then it is an "unused" field. If the unused field is an output field, then of course there is nothing unusual. If the unused field is a field in a source file, then a warning is sent to indicate that the field is not used in any assertion (Message 5). If the unused field is an interim field then the digraph is incomplete since there is no assertion involving the field, and an error message is sent to this effect (Message 5).

(b) If the node, say j, corresponding to a "keyed" input record has no "pointing" source, (i.e. an ISAM file that has no assertion "pointing" to its records)

$(\nexists i)(M_{ij}=5)$

then there is no assertion telling how that file relates to other files. The digraph is thus disconnected and therefore incomplete. In such a case, the user is warned that the two or more source file are defined but that there is no relation between the two (Message 8).

(c) If a field, j, has more than one assertion as its source, i.e. there exist k and l such that $M_{kj}=M_{lj}=3$, then a warning message is sent to the user indicating that the two assertions possibly present a contradiction. In such a case, the two assertions can only hold if they are under mutually exclusive choices, and a corresponding message is sent to the user (Message 9).

(d) Another check that needs to be made is that the targets of all assertions may not themselves be a field in a source file; i.e. if Aij=3 where i corresponds to an assertion, then j may not correspond to a field in a source file (Message 12).

Note that if any errors have been detected during the construction or during the post-analysis of the weighted adjacency matrix, the error count flags the Processor not to proceed to subsequent phases, but to let the user resubmit a corrected specification.

4.3.3.9 Path Matrix Creation; Cycle Detection and Enumeration

Another important type of analysis performed here is the detection and enumeration of any cycles that might exist in the digraph. This is necessary to give the MODEL user feedback about possible errors regarding circular definitions.

In order to perform such analysis, the <u>path matrix</u> (or <u>reachability matrix</u>) of the digraph is generated first. The path matrix, P, is a matrix of ones and zeros with a "1" in row i and column j if and only if there is path of any length from node i to node j. Formally,

Pij=1 if there exist k1,k2,...,km such that

A(i,k1)=A(k1,k2)=...=A(km,j)=1

Pij=0 otherwise

In other words, the path matrix indicates which nodes can be reached from other nodes. The path matrix can be defined as

$A \vee A^2 \vee \dots \vee A^n$

(sometimes called the "transitive closure" of A), but a more efficient method for generating it from the Adjacency Matrix, A, is Warshall's algorithm [WAR68], implemented in the CRPATHS (CReate PATH matrix) subroutine. Algorithm CRPATHS shows this procedure:

Algorithm CRPATHS: Create Path Matrix:

```
1. Let P=A (for all i,j)
2. Set j=1
3. Set i=1
4. If Pij=1 then set Pik=Pik∨Pjk (for all k=1 to n)
5. Set i=i+1; if i<=n, then go to 4.
6. Set j=j+1; if j<=n, then go to 3; else return.
```

Once the path matrix is created, the presence of one or more cycles is detected easily by searching for a "1" on the diagonal; i.e. (∃i)(Pii=1). If there are no cycles in the graph, the system proceeds to the next phase (precedence determination). Otherwise, we need to enumerate exactly which nodes appear in which cycles, information which the path matrix itself does not provide (a 1 on P's diagonal only tells us that the node is on some cycle). It is necessary in such cases to enumerate to the user the exact distinct sets of statements with circular definitions. Algorithm CYCLES is for enumerating all distinct cycles in the represented digraph (given the adjacency and path matrices). It is adapted from [BER 71] which in turn is based on Floyd [FLO67].

The algorithm has 3 inputs: the number of nodes, n; the Adjacency Matrix, A; and the Path Matrix, P. The algorithm finds all the cycles by the principle that node i is in a cycle with node k, if Aik x Pki = 1; i.e. there is an arrow from node i to node k and a path from k back to i; i.e. there is a cycle (i,k,...,i). If, however, Aik x Pki = 0

Algorithm CYCLES: Cycle Enumeration

```
Step 1. Root=1.

Step 2. (initiate tree; steps 2 to 6):
        Set REACHJ (k) = Root (for k=Root to n)
Step 3. Set USED (k) = 0 (for k=Root to n)
Step 4. Set level=1.
Step 5. PATH (1)=Root
Step 6. Set i=Root.

Step  7.  (Test if current path can be extended with nodes in a cycle;
Steps 7-11):
        If REACHJ (i)>n then go to Step 12.
Step 8. Set j= REACHJ (i).
Step 9. If A(i,j)*P(j,Root)=1 and ~ USED (j) then go to Step 18.
Step 10. Set j=j+1.
Step 11. If j<=n then go to Step 9.

Step 12. (Backtrack in tree, resetting REACHJ and USED ;
        Steps 12 through 17):
        Set REACHJ (i)= Root.
Step 13. Set USED (i) = 0.
Step 14. Set level=level-1.
Step 15. If level=0 then go to Step 26.
Step 16. Set i = PATH (level).
Step 17. Go to Step 7.

Step 18. (Extend path; Steps 18 through 23):
        Set USED (j) = 1.
Step 19. Set REACHJ (i) = j+1
Step 20. Set level=level - 1.
Step 21. Set PATH (level) = j.
Step 22. Set i=j.
Step 23. If j ~=Root then go to Step 7.

Step 24. (Print Cyclic Path):
        Print PATH (k), k = 1 to level (message #7).
Step 25. Go to Step 13.
Step 26. Set Root=Root+1.
Step 27. If Root <= n then go to Step 2.
Step 28. Return.
```

for all k, then node i does not lie on any cycle. The algorithm determines the cycles by growing all the trees such that Aik x Pki = 1 and the nodes of each path of the tree constitute the members of the cycle.

Algorithm CYCLES is best understood with an example. Figure 4.12 illustrates a small digraph with all the trees constructed by the algorithm. This example is taken from [BER 71]. Each path from the root of the tree to the same terminal node represents a cycle.

Note that the order of cycles printed by the algorithm is by lexicographic order of the node numbers. Since the corresponding dictionary has been previously alphabetized, the algorithm prints the distinct cycles in alphabetical order.

However, there are some situations which the system currently does not check. These deal with special MODEL names such as POINTER, EXIST, etc. As mentioned in Chapter 3, the user should never define LEN.X or EXIST.X in terms of X since there is an implicit precedence of LEN.X or EXIST.X before X. If the user did this, the result would be a type of cycle which this algorithm is not designed to detect.

An example of an illegal cycle in a MODEL digraph would be a set of circular assertions such as the following:

A=B+C

B=C+D

D=C+A

In this example, A depends on B, B depends on D, and D depends on A,

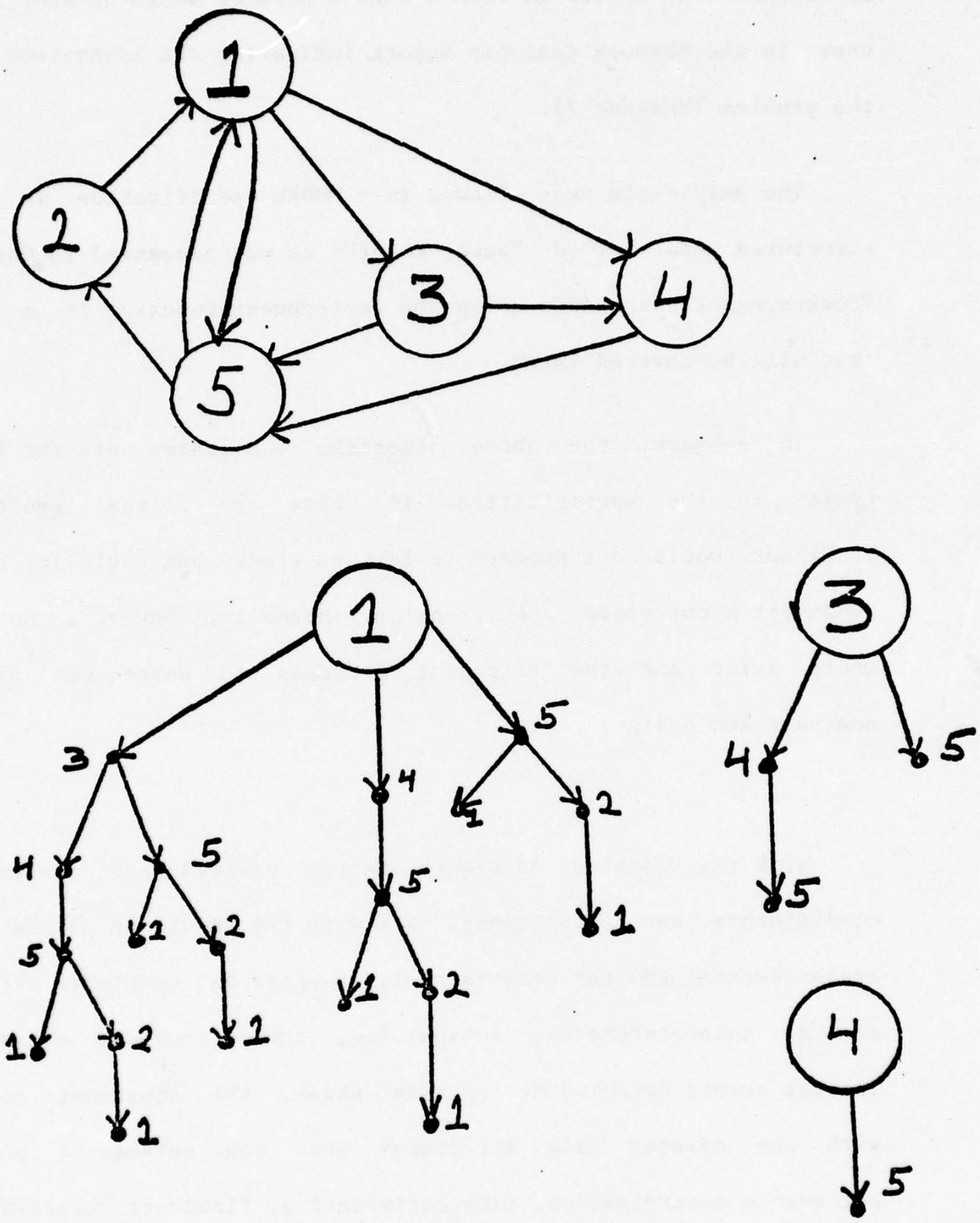


Figure 4.12 Cycle Enumeration of a Sample Digraph

an inconsistent cycle. In such a case a message would be sent to the user in the Network Analysis Report indicating the assertions causing the problem (Message 7).

The only cycle ever allowed in a MODEL specification is a very structured one called "replacement" as was presented in Chapter 3. Processing of assertions using the replacement function is a subject that will be covered later.

In summary, the above algorithm enumerates all the distinct cycles in the specification. If there are illegal cycles, the Processor would not proceed to further stages but would let the user re-submit a corrected specification. Normally, however, no cycles would exist and the Processor proceeds to subsequent phases of analysis and design.

With the Weighted Adjacency Matrix created and analyzed for completeness and consistency, and with the existence of any illegal cycles determined, the network analysis phase is complete. If there are no inconsistencies, ambiguities, incompleteness, or any other logical errors detected during this phase, the Processor continues with the created data structures into the subsequent phases of precedence determination, flow optimization, flowchart creation, and code-generation. Otherwise, the MODEL user has been presented with a set of reports pinpointing the causes and nature of his problem, and suggestions for rectifying them.

4.4 Automatic Program Design and Determination of Sequence and Control Logic

This phase involves ordering of all the events as implied by the precedence graph (represented by the weighted adjacency matrix) and determining the sequence and control logic. Program design proceeds with analysis of scopes and iterations, and with optimization of the flow. The result of this phase is a flowchart-like set of data structures (and a report) embodying the sequence and control logic.

4.4.1 Precedence Determination and Sequencing Algorithm

Once construction and above analysis of the matrix is complete, the next task is to analyze the matrices for the purpose of determining the sequences of events according to precedence.

The basic task of this algorithm is to take a given n x n adjacency matrix, A, such as in Figure 4.9 corresponding to the digraph of Figure 4.8, to rank the nodes according to precedence, and to reorder the nodes according to their rank.

It is known from graph theory (see such textbooks as [BER71]), that a given adjacency matrix A, gives all the paths of length 1 from i to j; A^2 gives all the paths of length 2; A^j gives all the paths of length j; etc. A multiplication of a matrix by itself here is boolean. Therefore one way of determining the precedence ordering of all the nodes is by successively multiplying the adjacency matrix A by itself. At each stage, the nodes of the current rank would be those with an all-zero column, because those nodes would have no predecessor with the current length path. Given a well-formed graph, this algorithm would terminate when all entries in A would be zero in at most n stages (j<=n).

This approach is inefficient for computer implementation, however, because at each stage, a matrix multiplication of an n x n matrix , A, by another n x n matrix, A^{j-1}, would be involved to get A^{j}. To produce the A matrix at each of the up to n stages involves n steps (n row-column matches for each of the n entries of the matrix) with the non-zero elements being multiplications. Thus, the entire process takes on the order of n steps.

A much quicker algorithm for determining the order of precedence is given in Algorithm PRECED. The explanations of each step are given in the table along with the algorithm for readability. This algorithm is similar to topological sorting algorithms such as the one found in [KAH62]. It involves analyzing only the original matrix without any multiplications and is accomplished in n stages, with each stage taking 2kn steps, where 1 <= k <=n. Thus the total precedence determination process that follows takes on the order of n steps.

The algorithm works by first finding all the nodes of rank 0; i.e. all the nodes which do not have precedents (Step 2). This is simply all the nodes which have all zeros in their column. (In Step 2, these are all column nodes j that are put in set D(0); the "i" in the condition are the row entries in each such column j). These nodes become the elements of rank set D(0), and the rank of all such nodes

Algorithm (PRECED): Precedence Determination
the following symbols are used:

A The input n x n adjacency matrix (row and column for each node)

i row index for A

j column index for A

D a vector of "rank sets"; each rank set (element of the vector) consists of a set of nodes at that rank; in the example in this section, D(0)={3,9,21}; D(1)={15,16}; etc.

l rank counter; index to D (i.e. in the algorithm, D(l) is the set of nodes of rank l; D(l-1) is the set of nodes of rank l-1, etc.)

n the number of nodes; also the number of rows and columns of A; also the number of elements in vectors R and O.

P is set successively to each node in the previous rank set, D(l-1); indexes row of A

q is set successively to each node in the current rank set D(l); indexes column of A; also indexes R

R the "rank vector" that is produced (has n elements); the index to R is a node number; the value of each element of R gives the rank of that node; e.g. R(q) gives the rank of node q.

O the "order vector" produced (has n elements); the indices to O are the sequence or step numbers (1,2,3,...); the value at each element of O is the node number to be executed at that position. In the example in this section, O(1)=9, O(2)=3, etc., meaning node 9 is fist, node 3 is second, etc.

(these are illustrated in an example following the algorithm)
is set to 0.

Secondly, the nodes of rank 1 are then all those nodes which are direct descendants of nodes in rank 0; the nodes of rank 2 are then all those nodes which depend on nodes in rank 1 (possibly updating the previous rank of some nodes); and so forth (Steps 3-6). At each stage, the algorithm has to check the rows of the previous set of nodes for direct descendants (Step 5). After the nodes have thereby been partitioned into <u>rank sets,</u> the order of execution of the nodes is

STEP	ALGORITHM	EXPLANATION
1	Initialize R to all zeros	Initially Rank vector all 0
2	$D_0 = \{ j \mid \forall i (A_{ij} = 0)\}$ If $D_0 = \emptyset$ then go to 9	Nodes of rank 0 consist of all those which have no precedents i.e. all 0 column
3	$l \leftarrow 0$	Index for rank set initially 0
4	$l \leftarrow l+1$ If $l = n$, go to 9	
5	$D_\ell = \bigcup_{p \in D_{\ell-1}} \{q \mid A[p,q]=1\}$	Next rank set consists of all those nodes which depend on something in the previous rank
6	$R_q \leftarrow \ell \quad (\forall q \in D_\ell)$	All nodes in current rank set are ranked with level 1
7	If $D_\ell \neq \emptyset$ then go to 4 otherwise go to step 8	If there are still nodes in current rank set, go back to find next rank set
8	Set Order vector to Rearranged nodes in Rank ascending order (normal exit)	Nodes are now ranked; simply rearrange nodes in rank order
9	There exists at least 1 one cycle somewhere in the digraph	This is because the algorithm has gone thru n rank sets and dependencies still exist on last one

Algorithm PRECED: Precedence Determination simply a re-arrangement of the nodes according to their rank (Step 8). The result of this algorithm is an order vector O, where O(i) is the node to be executed at step i.

The algorithm terminates when either all nodes have been ordered or, theoretically, if a cycle exists in a network (rank >n). The latter is impossible, however, because any cycles would have been detected in the earlier cycle detection and enumeration algorithm, and the Processor would not have reached this point. This algorithm operates on a non-cyclic digraph and orders all nodes.

In order to illustrate the algorithm, it is applied to the Adjacency Matrix of Figure 4.11, which, in turn, corresponds to the digraph and example of Figures 4.9 and 4.10. The rank sets and rank vector produced for this example are shown below, while the nodes sequenced in precedence order as a result are shown in Figure 4.13.

Rank Sets

D(0)={3,9,21}

D(1)={15,16}

D(2)={22,23,24

D(3)={19,20,26}

D(4)={14}

D(5)={13}

D(6)={10,11,12}

D(7)={1,6,7,27}

D(8)={5,18}

D(9)={8,25}

D(10)={4,17}

D(11)={2}

Rank Vector

(1)	7
(2)	11
(3)	0
(4)	10
(5)	8
(6)	7
(7)	7
(8)	9
(9)	0
(10)	6
(11)	6
(12)	6
(13)	5
(14)	4
(15)	1
(16)	1
(17)	10
(18)	8
(19)	3
(20)	3
(21)	0
(22)	2
(23)	2
(24)	2
(25)	9
(26)	3
(27)	7

SEQUENCE OF PROCESSING

ORDER			
VECT.	ORDER	NAME	RANK
INDEX	VECTOR		
1	3	MINSALE	0
2	9	OLD.INVEN	0
3	21	SALETRAN	0
4	15	SALEDECK	1
5	16	SALEREC	1
6	22	SALETRAN.CUST#	2
7	23	SALETRAN.QUANTITY	2
8	24	SALETRAN.STOCK#	2
9	19	SALESLIP.CUST#	3
10	20	SALESLIP.STOCK#	3
11	26	TRINV	3
12	14	POINTER.OLD.INVREC	4
13	13	OLD.INVREC	5
14	10	OLD.INVEN.QOH	6
15	11	OLD.INVEN.SALPRICE	6
16	12	OLD.INVEN.STOCK#	6
17	1	CALCCHRG	7
18	6	NEW.INVEN.SALPRICE	7
19	7	NEW.INVEN.STOCK#	7
20	27	UPDQUAN	7
21	5	NEW.INVEN.QOH	8
22	18	SALESLIP.CHARGE	8
23	8	NEW.INVREC	9
24	25	SLIPREC	9
25	4	NEW.INVEN	10
26	17	SALESLIP	10
27	2	INVDISK	11

Figure 4.13

Digraph Nodes Sequenced in Precedence Order

At this stage, the Processor does not show the detailed tasks taking place at each node or the control logic. Those details are generated later in the "Flowchart Table Generation" phase (Section 4.4.3). Some of these nodes will correspond to code to be generated. Notice that the order that could be generated here is not unique since nodes of the same rank set could come in any order. Also, compare the generated order of Figure 4.13 with the digraph of Figure 4.10 to observe the sequence of events.

The generated sequence of nodes, or order vector is the backbone of the sequence and control logic design. It is subsequently used in the "flowchart" and code generation phases which generate code corresponding to each node, a subject to be dealt with later.

Another by-product of this algorithm is the identification of events that can occur in parallel; i.e. those of the same rank set. For example, nodes 23 and 24 (NEW.INVREC and SLIPREC) could be executed in parallel since they are in the same rank set (rank 9). In this example, they are both records so the WRITE statements to be generated later could theoretically be executed in parallel. This could become important in the future for system-generated code for a parallel processor, and appears later for the user in the Flowchart Report.

4.4.2 Scope and Iteration Analysis

This routine (FLOWOPT) has the tasks of (1) determining the scopes of the iterations described in the specification by use of the FOREACH option for repeating groups or fields; (2) ensuring that the events taking place within each iteration are only those that are necessary; and (3) re-ordering those events not involved in the loop. Algorithm FLOWOPT shows the steps of this process. It involves analyzing the order vector produced in the previous section. Steps 2 through 6 are concerned with finding the beginning of a loop. The first assertion using a FOREACH name (as either source or target) that has not yet been entered in the list-of-loops including this node begins a loop. Therefore each assertion heading is examined for every source or target name to check if it uses a FOREACH that has not been accounted for yet at the current node (Steps 3-6). The last node in the loop (El) is found by looking for the furthest node in the order vector that also uses the current FOREACH name (Step 7). The position of the last node of the loop is also saved in Step 8 because it might change as some nodes are moved beyond the end of the loop later (i.e. El may change). Steps 9 through 15 deal with analyzing the nodes in the scope and determine which nodes in between are to be included in the scope. Each node, beginning with the start node (Step 9) and up to the ending node (Step 16), is analyzed as to whether or not it belongs in the loop. At this point, nodes that do not belong in the scope may well be caught in the middle of this range simply because of their ranking. The nodes to be included in the scope of the iteration are only those nodes which are descendants of nodes above them in the

Algorithm FLOWOPT: Flowchart Scope Analysis and Optimization

The following symbols are used:

O the input order vector as defined in Algorithm PRECED

i index to O; O(i) is the node number at position i

n the number of elements of vector O, which is also the number of rows and columns of P

P the path matrix as defined in Section 4.3.3.9 (the row and column indices are node numbers)

S1 position of the starting node of current loop

E1 position of the ending node of the current loop

l index to O to find end of loop

B position of the last node that has been moved out of and beyond end of loop

k is set successively to nodes in the order vector O between S1 and E1

S2 starting position of each other loop in succession used in checking consistency

E2 ending position of each other loop in succession used in checking consistency

j index to O (between S1 and k)

CURRENT-FOREACH
name of current FOREACH variable

LIST-OF-WRITES-TO-BE-MOVED-DOWN
list of names of records and files to be moved beyond end of loop

NODES-MOVED-TO-TOP-FLAG
flag set to 0 before any nodes are moved in front of start of current loop; set to 1 after a node is moved to the top

LIST-OF-LOOPS
list of data about each loop as follows:

START-OF-LOOP END-OF-LOOP FOREACH-NAME

Algorithm FLOWOPT: Flowchart Scope Analysis and Optimization

Step 1. Set i=1 (start with first node)

(Finding beginning of next loop, Steps 2-6):

Step 2. If node O(i) corresponds to an assertion then go to Step 3; else go to Step 20.

Step 3. If assertion uses any more fields with a FOREACH name (as either source or target) then go to Step 4; else go to Step 20.

Step 4. Set CURRENT-FOREACH =name of this FOREACH variable

Step 5. If this loop has already been considered (i.e. there is a FOREACH name in the LIST-OF-LOOPS = CURRENT-FOREACH & START-OF-LOOP <=i<= END-OF-LOOP in the LIST-OF-LOOPS entry)
then go to Step 3.

Step 6. Set S1=i (start of current loop).

Step 7. (Finding end of loop):
Set E1=largest l such that node O(l) is an assertion with CURRENT-FOREACH

Step 8. Set B=E1 (initial position of last node that has been moved out of and beyond end of loop)

(Analyzing scope, Steps 9-16):

Step 9. Set k=S1+1 (begin with next node in loop after S1)

Step 10. Set MOVED-NODES-TO-TOP-FLAG =0
Clear LIST-OF-WRITES-TO-BE-MOVED-DOWN.
(initially no nodes have been moved before or after the loop)

Step 11. If node O(k) is a descendant of any node between S1 and k (i.e. there is a j, S1<=j<k such that P[O(j),O(k)]=1)
OR if node O(k) is an assertion using CURRENT-FOREACH (as either source or target)
then node O(k) belongs in loop, go to Step 14.

Step 12. (Node O(k) does not belong in loop and needs to be moved in front of loop).
Move node O(k) to position before S1 by switching.
Set S1=S1+1 (so that it still points to beginning of loop).

Step 13. Set MOVED-NODES-TO-TOP-FLAG=1

Step 14. If node O(k) is a repeating group or field in a target record that is using CURRENT-FOREACH , then retrieve its parent record & file names & save in LIST-OF-WRITES-TO-BE-MOVED-DOWN.

Step 15. If node O(k) is in LIST-OF-WRITES-TO-BE-MOVED-DOWN then move node O(k) after the end of the loop (by switching node O(k) up to position B, and setting E1=E1-1 so that it still points to end of loop not including the moved nodes).

Step 16. (look at next node in scope):
Set k=k+1
If k>E1 then go to Step 17; else go to Step 11.

Step 17. Add entry to LIST-OF-LOOPS:
Set START-OF-LOOP=S1
Set END-OF-LOOP=E1
Set FOREACH-NAME=CURRENT-FOREACH

Step 18. Set S2= START-OF-LOOP of each other loop, in succession, in LIST-OF-LOOPS
Set E2= END-OF-LOOP of each other loop, in succession, in LIST-OF-LOOPS

Check consistency of scope of current loop with respect to others; check that one of the following must hold; if not, then error.

S1<=S2<=E2<=E1 or
S2<=S1<=E1<=E2 or
S1<=E1<S2<=E2 or
S2<=E2<S1<=E1.

Step 19. If any nodes were moved before position S1 (i.e. MOVED-NODES-TO-TOP-FLAG=1) then go to Step 2 (the scan for next loop will start at i to check for possible other loops in the nodes moved to the top); otherwise go to step 3 (to check if node O(i) begins any other new loop).

Step 20. (Get next node in O):
Set i=i+1
If i>n then go to Step 21; else go to Step 2.

Step 21. Return.

loop or other nodes which also use the current FOREACH (Step 14). For the dual purpose of untangling the scopes and for optimization, only such nodes that belong in the scope of the iteration are left in the loop by moving out those that do not belong (Step 12).

When nodes are moved to the point immediately preceding the start node, they keep their relative position to each other. They are said to be invariant to the iteration since they do not depend on the repetition.* In such cases of nodes moved to the top, a flag is set (Step 13) because when this loop is finished being analyzed later (Step 19), the scan for other loops will be resumed starting with these moved nodes. This was also one of the reasons for Step 5 to ensure that each distinct loop is analyzed once.

It should be noted that when records are caught in the scope of the loop, sometimes the associated input/output command remains in the loop, wherease at other times it should not. For example, a record having a pointer to it by means of a POINTER type assertion using the current FOREACH depends implicitly on the same FOREACH itself; it therefore should be in the loop. This is accomplished by the condition in Step 11. However, target records containing a repeating group or field using the current FOREACH name could conceivably be caught in the middle of the loop due to ranking, but they should be moved beyond the end of the loop because the iteration is "filling up" the

* The PL/1 Optimizing Compiler also removes invariant operations out of an iteration, but its optimization is not as general or as complex, because the scopes of the iterations are pre-designated explicitly for it by the programmer's control code (DO and matching END statements).

repeating group or field, and the record need not be written until it is complete. Therefore such records are moved beyond the end of the loop (Steps 14-15).

The algorithm goes on to examine each node between the start node and the end node successively (Step 16) to check whether or not it belongs in the loop, until the ending point, El, is reached.

When the scope and constituents of the loop are finished being determined, the information is put into the LIST-OF-LOOPS (Step 17).

Another step in this procedure is to ensure that the different iteration scopes implied in the specification, if there is more than one, have consistent scopes (Step 18). Each distinct pair of iterations must be either disjoint or nested. That is, if Il=(sl,el) and I2=(s2,e2) represent the iteration scopes of iterations Il and I2 (with s=starting node and e=ending node), then one of the following must hold true for the scopes to be meaningful:

(1) (a) sl<=s2<=e2<=el (i.e. I2 nested in Il);

(b) s2<=sl<=el<=e2 (i.e. Il nested in I2);

(2) (a) sl<=el<s2<=e2 (i.e. Il disjoint and precedes I2);

(b) s2<=e2<sl<=el (i.e. I2 disjoint and precedes Il).

If the scopes are otherwise "tangled", a message is sent

ERROR(INCONSISTENCY): FOREACH x AND FOREACH y HAVE INCONSISTENT SCOPES

The final output of this procedure is

(1) an updated order vector;

(2) an updated rank vector;

(3) a list of iteration scopes with the following information for each scope (sorted on the index to the order vector):

(a) starting node;

(b) ending node;

(c) the governing iteration variable (FOREACH name).

These lists (DOTAB and ENDTAB) are used later by the "flowchart generation procedure" to generate the equivalent of PL/1 iterative statements ("DO-loops").

Many kinds of optimization can be performed both locally and globally. However, these are left to the compiler which will eventually receive the generated program because they are known technology.

4.4.3 Generation of Flowchart Table

This sub-phase is concerned with the generation of a conceptual "flowchart" of the desired object module, independent of the object programming language, in the form of a table, from which generation of two products would subsequently follow easily:

(1) the object PL/1 program; and

(2) a flowchart-like report.

The previous phase, which analyzes the network relationships (Section 4.3), and the first two sub-sections of this phase, which determine precedence (Section 4.4.1) and iterations (Section 4.4.2), resulted in a set of data structures: the adjacency matrix, path matrix, order vector, and iteration list. The Processor now continues t build the "flowchart" from these data structures. Recall that this entire phase (Section 4.4) would not be reached had there been user errors detected during network analysis (Section 4.3).

In a sense, the order vector that was generated during the precedence determination phase already presents a skeleton flowchart of the object program, because it defines the names of each node in the order that the corresponding event is to be executed. For example, if a node of the order vector is the name of a record which is in a source file, it would correspond to a READ statement (among other code associated with input/output) to be executed at that point. In order to complete the list of names in the order vector into a flowchart and then a program however, we need to get more information about each node to the extent that the PL/1 code necessary to effect the desired operation could then be generated. Returning to the same example, if a node in the order vector represents the name of an input record, we need such additional information as the length of the record, whether it has fixed or variable length, the name of the associated file, the file organization, etc. before we could generate the corresponding PL/1 READ statement. All such needed information about a given node, however, is very readily available by use of the retrieval sub-system that was described in Section 4.2.3, capitalizing on the internal associative organization of the user specification. For example, in order to retrieve the storage entry on the file in which the record name is contained, we issue the following:

```
CALL RETRIEVE (RECNAME ||'&$FILE',...);
```

Thus, creating the flowchart of the desired program module involves linearly traversing the order vector, and for each node, determining the type of operation that corresponds and retrieving all the relevant information needed in order to generate the necessary corresponding PL/1 statement or statements subsequently.

As Figure 4.14 indicates, we have a two step process:

(1) generating a table of events or flowchart; and

(2) generating the corresponding PL/1 code.

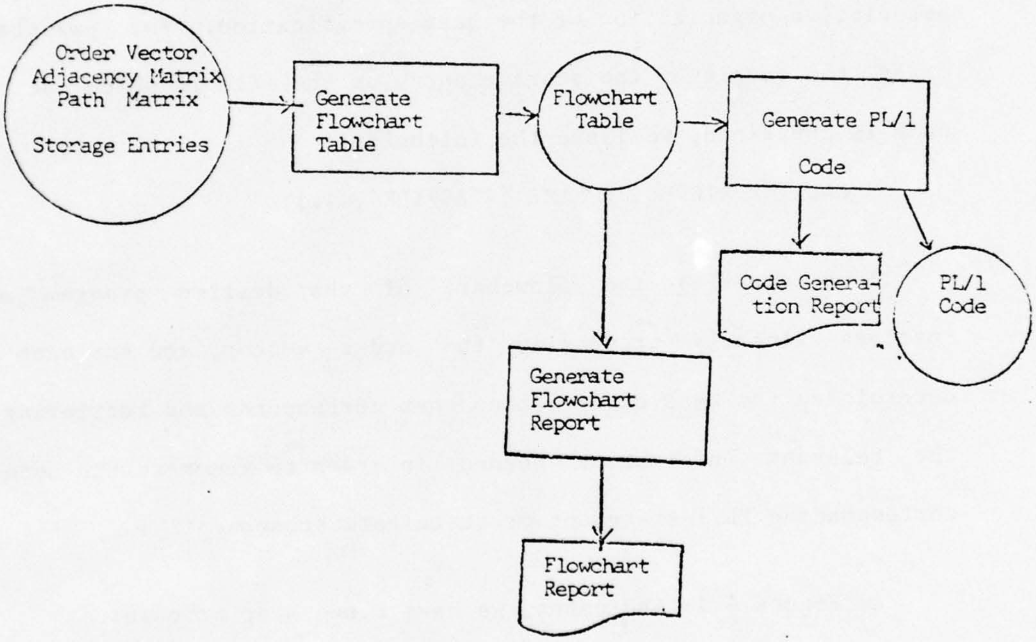


Figure 4.14

Generating Object Program after Analysis and Precedence Phases

The fact that PL/1 code is not generated directly and that the intermediate flowchart table is created has many justifications. It makes the program generation process more modular. The intermediate language-independent table, which contains the type of operation and associated information at each step, can be examined without involvement in the intricacies and syntax of the object PL/1 language. In fact, as just shown in the figure, a pictorial version of the flowchart, to be described later, could be generated from the table and be useful to the systems programmer. Moreover, the entries in the flowchart table are language independent to the extent that code could be generated for another language such as COBOL, changing only the code generation portion that follows flowchart generation.

4.4.3.1 Control Module for Generating Flowchart Table

This module of the Processor traverses the nodes of the order vector, one at a time, determines the type of node, and calls the appropriate subroutine which creates an entry of that type in the flowchart table. Each entry in the flowchart table to be generated has the following general form:

Node#	Node type	Name	Operation & auxiliary information

where the node# is the index to the dictionary entry (the dictionary of names described in Section 4.3.3.1) with the node's name (the current node being traversed), for which code is being generated; the node type is an abbreviation indicating the kind of name to which the node corresponds (for example, record, field, assertion); and the operation and auxiliary information contain all that is needed to generate code for that node (for example, READ and its parameters).

Algorithm GENFLT shows the Generate Flowchart procedure. the subroutines, which are called on the basis of the node type (Step 5), will have the task of filling in this information for the particular kind of operation. The operation and auxiliary information for each node type is described in each subsequent section. Figure 4.15 shows the components of generating the flowchart table presented as routines in the sections to follow. These include identifying input/output commands (IDIOCD), identifying assertions information (IDASSN), identifying field associations (IDFLDAS), and generating a table of declarations (GDCLT). The control routine also calls routines (CHECKDO and CHECEND in Steps 2 and 6) to check the iteration table upon each step through the order vector for possible generation of iterative control structures (for "DO-loop and END statement in PL/1). Furthermore, it calls a routine to check for generation of statement labels when necessary (Step 4, CHECLAB). At the end of looping through the order vector and invoking the corresponding flowchart generation

Algorithm GENFLT: Generate Flowchart Table

[Subroutines called: CHECKDO, CHECOND, CHECLAB, CHECEND, IDASSN, IDIOCD, IDFLDAS, IDMODNM]

Step 1. Set i=1.

Step 2. Call CHECKDO (check for iterations).

Step 3. Call CHECOND (check for conditionals).

Step 4. Call CHECLAB (check for labels).

Step 5. Branch on Node Type of O(i) (the "Order Vector" as described in Section 4.4.1):

If Node Type= ASSN then call IDASSN (identifying assertions information).

If Node Type= RECD or RPTN then Call IDIOCD (identifying i/o commands).

If Node Type= FLD or INTR then Call IDFLDAS (identifying field associations).

If Node Type= MODL then Call IDMODNM (identifying module name).

If other type, then create dummy flowchart table entry.

Step 6. Call CHECEND (Check for end of iterations).

Step 7. Set i=i+1.

Step 8. If i<=n then go to Step 2.

Step 9. Call IDRSET (reset housekeeping variables).
Call IDGOTO (generate branch to next read).
Call IDFIN and IDEND (end-of-program wrapup tasks).

Step 10. Return.

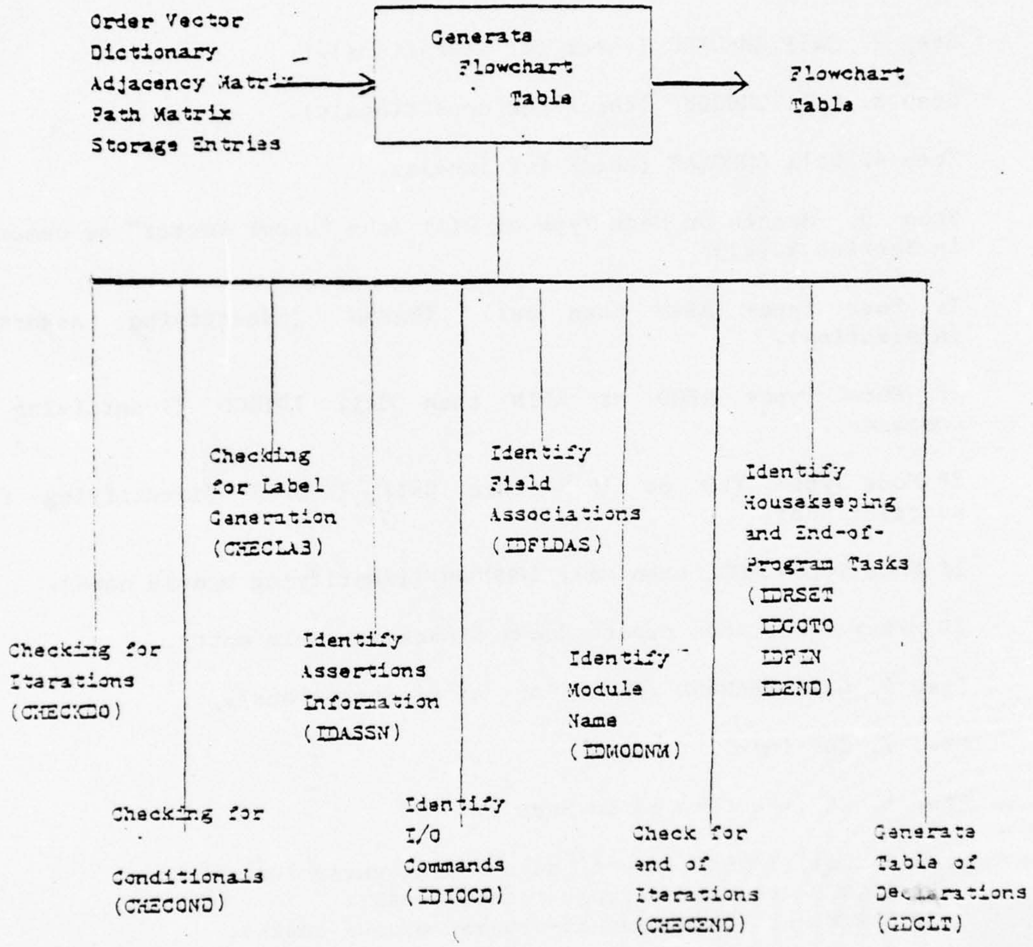


Figure 4.15

Components of Generating the Flowchart Table routine, subroutines are called (Step 9) to identify "housekeeping" variables that need to be reset (IDRSET); to identify a branch to read the next record (GENGOTO); to identify the end of program wrapup tasks (IDFIN and IDEND). All these subroutines are described in the subsections that follow.

4.4.3.2 Identifying the Module Name

The routine to "Identify Module Name" (IDMODNM) is a trivial one whose task is to retrieve the module name and create a flowchart entry for it. This routine is invoked from the "Generate Flowchart Table" routine when it detects the name of a module, which should be the first entry of the order vector. This routine simply creates an entry in the flowchart table in the following form:

NODE#	NODE TYPE	MODULE NAME

where the NODE# is the number of the node as given by the order vector; NODETYPE is set to 'MODL' to indicate that the node is for a module statement; and the MODULE NAME is as given. This flowchart table entry will be used later by the corresponding code-generation routine to generate the PL/1 procedure declaration.

An example of such an entry from the module statement of the MINSALE problem presented earlier in this chapter is the following:

3	MODL	MINSALE

4.4.3.3 Identifying Input/Output Commands

The routine "Identify Input/Output Commands" (IDIOCD) has the task of collecting all the information necessary for the writing of an input/output statement in the final PL/1 program. This routine is invoked from the Generate Flowchart Table (GENFLT) procedure described in the previous section, upon finding a node with a record name. It has, as input, the dictionary entry of the record name for which code is being generated. The generated input/output operations and information that will be needed later to generate the PL/1 code is put into an entry of the flowchart table (FLOWTAB_REC) which has the following format:

NODE#	NODETYPE	RECNM	IOMODE	FLNAME	ORG	KEYED	KEYNM	PACKINF

This information has the meaning explained below, and is used during the code generation phase to produce i/o operations. The components of this flowchart table entry are either given to this procedure or the information is readily available from the stored statements via the RETRIEVE system.

Algorithm IDIOCD describes the "Identifying I/O Commands" procedure, and how it creates this flowchart table entry.

NODE# is the entry number within the dictionary containing the record name, for which code is being generated. This is passed from the calling program (Step 1).

NODE TYPE is set to 'RECD' to identify that this entry is an input/output operation for a record (Step 2).

RECNM is the name of the record for which the input/output operation is being generated. It comes directly from the dictionary entry given to this procedure (Step 3).

IOMODE is set to 'RD', 'WR', or 'RW' by this procedure in order to indicate whether the generated command is to be a 'READ', 'WRITE', or 'REWRITE', respectively (Step 9); this is determined according to whether the file is source or target and whether the file organization is sequential or indexed according to the chart below. The information is available by retrieval.

Algorithm IDIOCD: Identify Input/Output Code

[Subroutines called: RETRIEVE]

Step 1. Set NODE# in flowchart table entry to current node#.

Step 2. Set NODETYPE to RECD .

Step 3. Set RECNM to name of record according to DICT (node#).

Step 4. RETRIEVE storage entry for record name.

Step 5. RETRIEVE corresponding file name, key name, and storage device.

Step 6. If file is keyed, set KEYED flag to 1; else set to 0.

Step 7. RETRIEVE file organization.

Step 8. Add a suffix to the file name (FLNAME) according to the "File Name Formation" chart shown and explained in the text below.

Step 9. Determine IOMODE as READ, WRITE, or REWRITE according to record use and file organization (see next chart in this section).

Step 10. If record is target and had a subset specification, then generate a conditional flowchart table entry to circumvent the WRITE (generates "IF ~SUBSET.filename").

Step 11. (see Algorithm IDPACK for details of this procedure). If record has any variability in lengths or repetitions, then create "packing" information in flowchart entry.

Step 12. Write flowchart table entry.

Step 13. Return.

"Input/Output Mode Determination" Chart

ORGANIZATION	RECORD USE	I/O MODE
Sequential	Source	Read
Sequential	Target	Write
Indexed	Source	Read
Indexed	Target	Rewrite

FLNAME is the name of the file from which the input/output operation is to be performed. It is directly retrievable from stored statements. In order to make the file name unique, however, a suffix indicating its use (source, target, or both) is added to it (Step 8), as shown in the chart below. The name formed here is the external file name that will be used to declare the file in PL/1 and will be referenced by all the generated input/output statements for that file. The main reason that the suffix is added is to distinguish it from the file name as given by the user which is reserved for the name of the highest level tree of an internal hierarchic buffer in the generated program (to be described later in Section 4.4.3.11). The reason the original file name, as given by the user, is used for the hierarchical structure is that it enables the user to qualify field names by referring to the files in which they are contained. For example, a user may refer to a STOCK# field in the INVEN file as INVEN.STOCK#, and therefore INVEN should be the name of the highest level in the internal tree-structured buffer.

"File Name Formation" Chart

ORGANIZATION	RECORD USE	UNIQUE FILE NAME
sequential	input only	filename||'S'
sequential	output only	filename||'T'
indexed	input only	filename||'S'
indexed	output only	filename||'T'
indexed	both I/O	filename||'U'

ORG is set to 'S' if the file is sequential or to 'I' if the file is indexed. This information is available from the storage description statements which are stored and retrievable by a call to the retrieve system (Step 7).

KEYED is a flag set to 1 if the file is "pointed to" or "KEYED ; i.e. if the key name in the stored statement is non-blank; it is set to 0 otherwise (Step 6).

KEYNM is the name of the field within the record serving as the key field. It is retrievable from the file description statement (Step 5).

PACKINF is information needed to generate code for data packing and unpacking when the variable-length and repetition features of MODEL are used. This is explained after the example below. The procedure creating it is given in Algorithm IDPACK and outlined in the next section.

An example of an entry in the flowchart table which corresponds to a record, from the MINSALE problem, is for SALEREC which is the name of the record for the sales transaction. The flowchart table entry for SALEREC would appear as follows:

Node#	Nodetype	Recnm	IOmode	Flname	Org	Keyed	Keynm	PackInf
5	RECD	SALE REC	RD	SALE TRAN	S	0	--	--

which indicates that SALEREC, the fifth ordered node, is the name of the record of the SALETRAN file, which is an input sequential file and from which SALEREC is to be read (RD).

4.4.3.4 Identifying Data Packing and Unpacking Information for Variable Repetition and Length

When all fields of a record are of a fixed length and occur a fixed number of times, code can easily be generated to have the read or write command transfer the data directly to or from the PL/1 hierarchic storage structure, which represents such data structures conveniently. The MODEL language, however, has facilities to describe variable-length fields or variably-existing groups or fields (via the LEN or EXIST assertion facilities) whereby the user can provide expressions to be evaluated at execution-time, which in turn determine the length, existence, or repetition of an item. Such general facilities for variability of data do not exist in PL/1 data structures. PL/1 does not have direct facilities, as does MODEL, to define the length of a field or the number of repetitions of a group or field by an aribtrary expression, such as by an arithmetic expression or by a function to scan for a delimeter. Also, in MODEL a field or group can be defined to be optional by describing it to occur a minimum of zero times with an associated EXIST assertion defining the number. There is no direct way to declare such a concept in PL/1. PL/1 provides a more limited variability capability with the REFER feature (by requiring the length or number of repetitions to be stored explicitly with the data in the record), and "variable-length" strings in PL/1 (which actually occupy the maximum length and store current length). Therefore, generation of input/output code for files with such variability cannot simply transfer the data directly in and out of the PL/1 data structure. Instead, the generated PL/1 code must simulate the variability provided by MODEL. Therefore, whenever a record described in MODEL has variable length or variably existing items (uses the MODEL LEN or EXIST assertions), extra code is going to be generated. The actual transformations from the MODEL data structures to those available in PL/1 will be shown in the corresponding code generation section (4.5.1.3).

For reading such variable records, the input buffer must be scanned, evaluating the LEN and EXIST assertions when they come up, and the data needs to be transferred field-by-field into the PL/1 structure. The information about variable data within the record, if any, that is necessary for such variable data "unpacking" operations is identified in the current flowchart table entry (in the "PackInf" segment above) for later code generation.

For writing such variable records, the data in the output PL/1 data structure must be extracted for the exact length or existence as evaluated by the assertions, and transferred to the output buffer. Such variable data "packing" information within the record are also identified here for later code generation.

Algorithm IDPACK shows the procedure for identifying the necessary information for later code generation. The data structures used and generated by this algorithm are discussed here. Identifying the variable information here involves a recursive routine to "climb" down the tree structure of the record and identify those items which have a variable length or existence. The algorithm stacks information in a temporary stack (TEMP-STACK) about each member of the record (field, group, and in turn its sub-members), as it climbs down the record tree-stucture (Step 2). The information about each member of

Algorithm IDPACK: Identify Data "Packing" and "Unpacking" Information for Variable Repetition and Length

Step 1. For each member (group or field) of the record, perform Step 2.

Step 2. Call CREATE-TEMP (member-name, #subs, subl,sub2) (a recursive procedure to enter packing information for all data items in the record into the flowchart table entry).

```
name -- name of group or field in record.
#subs -- 0 subscripts means no repetition;
         1 subscript means fixed number of repetitions;
         2 subscripts, variable no. of repetitions.
subl -- minimum number of repetitions
sub2 -- maximum number of repetitions
        (subl=sub2 implies fixed number of repetition).
```

Step 3. Copy TEMP-STACK (created by CREATE-TEMP subroutine) into the PACKINF segment of flowchart table entry.

Step 4. Return.

Algorithm IDPACK (continued): Subroutine CREATE-TEMP

(Subroutine to create information for each sub-member (group or field) of the record to be entered in the PACKINF segment of the flowchart table entry)

Step 1. Set NAME, #SUBSCRIPTS, SUB1, SUB2 (to become part of the PACKINF segment of the flowchart table entry) according to passed parameters.

Step 2. If member is a field, then go to Step 3; else go to Step 12.

Step 3. Set TYPE ='F' (field).

Step 4. Set ARITY =0 (field has no sub-members).

Step 5. RETRIEVE field storage entry.

Step 6. Set FIELD-TYPE to 'C', 'B', 'F', or 'N' according to whether field is character, binary, fixed decimal, or numeric character, respectively.

Step 7. If field is fixed-length, then set FIELD-LEN-TYPE to 'F'; else set it to 'V' (variable-length).

Step 8. Set MIN-LENGTH and MAX-LENGTH according to field storage entry.

Step 9. If field repeats variable number times (#SUBS=2), RETRIEVE EXIST assertion name evaluating repetitions & save in EXIST-PROC.

Step 10. If field is of variable length, RETRIEVE name of LEN assertion that evaluates it, and set LEN-PROC to it.

Step 11. Go to Step 15.

Step 12. (member is a group): RETRIEVE group storage entry;
If group repeats a variable number of times (#subs=2) then RETRIEVE name of the EXIST assertion that evaluates repetition and save in EXIST-PROC.

Step 13. Set ARITY = number of sub-members of group; fill in SUB1 and SUB2 (minimum and maximum repetitions).

Step 14. For each sub-member of group, call CREATE-TEMP recursively with parameters (name, #subs, sub1, sub2), stacking the above information for each level in the record tree-structure onto TEMP-STACK .

Step 15. Return.

AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416

UNCLASSIFIED 76-05 NL

4 OF 6
AD
A032638

the record in TEMP-STACK is put into the PACKINF portion of the "Record" flowchart table entry depicted above (Step 3). The PACKINF segment of the flowchart table entry has the following information for each member (group or field) of the variable record, created by the CREATE-TEMP subroutine in the algorithm. The CREATE-TEMP subroutine fills in this information for each member, be it a field (Step 3-11) or a group (Steps 12-14). In the case of groups, the subroutine is called recursively for its sub-members.

NAME -- name of member (group or field)

TYPE -- 'F' for field; 'G' for group.

#SUBSCRIPTS -- 0=no repetition; 1=fixed number of repetitions; 2= variable number of repetitions.

SUB1 -- minimum number of repetitions.

SUB2 -- maximum number of repetitions (for fixed repetition, SUB1=SUB2).

EXIST-PROC -- name of EXIST assertion evaluating number of repetitions.

ARITY -- number of sub-members.

FIELD-TYPE -- C for character; B for binary; F for fixed decimal; N for numeric character.

FIELD-LEN-TYPE -- F for fixed; V for variable.

MIN-LENGTH -- minimum length of field.

MAX-LENGTH -- maximum length of field (MIN-LENGTH=MAX-LENGTH for fixed length).

LEN-PROC -- name of LEN assertion evaluating length of field.

From this information, code can be generated later to transfer the "variable" data in and out of the object PL/1 hierarchical data structure (see corresponding sections in code generation on i/o and variability).

An example of an entry generated in the flowchart table for the variable-length record SALEREC from the DEPSALE problem is the following:

Node#	Nodetype	Recnm	IOmode	Flname	Org	Keyed	Keynm	PackInf
113	RECD	SALEREC	RD	TRANS	S	0	--	(see below)

"PACKINF" (Packing Information) segment example:

Name	Type	#	Sub	Sub	EXIST	Arity	FLD	FLD	MIN	MAX	LEN
		Subs	1	2	PROC		TYP	LEN	LEN	LEN	PROC
TERM#	F	--	--	--	--	0	C	F	5	5	--
CUST#	F	--	--	--	--	0	C	F	4	4	--
ACTCODE	F	--	--	--	--	0	C	F	1	1	--
CLERK#	F	--	--	--	--	0	C	F	1	1	--
DEPT#	F	--	--	--	--	0	C	F	2	2	--
TAXCODE	F	--	--	--	--	0	C	F	1	1	--
TRITEM	G	2	1	9	NTR	2	--	--	--	--	--
STOCK#	F	--	--	--	--	0	C	F	3	3	--
QUANTITY	F	--	--	--	--	0	N	F	2	2	--
ENDITEMS	F	--	--	--	--	0	C	F	1	1	--

4.4.3.5 Identifying Assertions Information

The routine "Identify Assertion Information" (IDASSN) has the task of collecting all the information necessary in order to generate later a PL/1 procedure to carry out the operations implied by the assertion and to generate a CALL to that procedure. This routine is invoked from the "Generate Flowchart Table" control routine (GENFLT) and has, as input, a dictionary entry of the assertion for which code is being generated.

Algorithm IDASSN presents the Identify Assertions procedure. It retrieves the text of the assertion (Step 2), collects it, and reformats it without extraneous spacing. Besides entering the name and text of the assertion in the flowchart table entry (Step 2), it also enters information for several special cases: if the assertion uses a function, it marks the use of the function in a table so that the function itself be included in the generated program (Steps 3-4). As explained further below, if the assertion uses a "conditional" function, the conditional function is entered in the "conditional list" so that conditional control code can later be generated (Step 6). Also explained further, if the assertion uses the "Replace" function, then the target of the replacement is entered in the flowchart table entry so that the later PL/1 code-generation routine will be able to generate replacement code (Step 5). Further comments on how code is generated for replacement are given below under the explanation of RPLAB and in the corresponding code generation section (4.5.1.7).

The language-independent information which is later used to generate the PL/1 procedure embodying the assertion and its invocation, and any necessary PL/1 control code to govern the procedure execution is put into an entry of the flowchart table (FLOWTAB_ASSN) which has the following format:

Algorithm IDASSN: Identify Assertion Information

[Subroutines called: RETRIEVE]

Step 1. RETRIEVE assertion storage entry.

Step 2. Fill in text, node#, type, and name of assertion in the flowchart table entry.

Step 3. If assertion uses a function, then go to Step 4; else go to Step 7.

Step 4. Mark use of function in USEFCN table (so that the function itself later will be included in the generated program).

Step 5. If assertion uses the Replace function, then save replacement target in the flowchart table entry.

Step 6. If assertion uses a conditional function, then add the assertion to the "conditional list" (used later to generate code to circumvent dependent nodes).

Step 7. If assertion specifies a source subset, then generate a flowchart table entry to go read the next record if the current record is not in the subset.

Step 8. Write flowchart table entry.

Step 10. Return.

NODE#	NODETYPE	ASSN NAME	FCN	RPLAB	TEXT

The information has the meaning given below and is readily retrievable from the stored assertion description.

NODE# is the number of the assertion node as given by the order vector.

NODETYPE is set to 'ASSN' in order to indicate the type of flowchart entry.

ASSN NAME is the name of the assertion for which code is being generated.

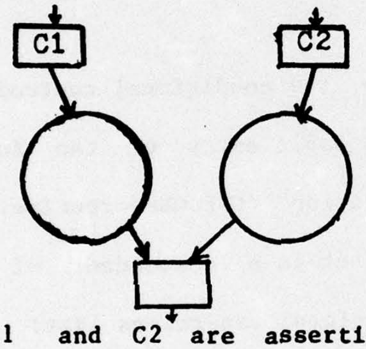
FCN is the name of a system-provided function, if any, that is used by the assertion. If the specification uses any system-provided function, the use of such a function is marked by the FCNUSED routine so that it will be included in the generated program later (by the MERGPL1 routine). Some functions provided by the system are not "completed" immediately upon invocation but only when an associated condition is met. In other words, some functions can have conditions associated with them which signal their completion. Such a mechanism turns out to be a useful and general facility. It applies only to certain "conditionally completed" system-provided functions to which the user has access in his assertion. For example, a summation function completes adding all the desired components only when it reaches the last desired item to be included in the total. The indication of the use of such a function in the flowchart entry here becomes necessary for the code-generation phase in order to generate conditional control code to test for the node's completion. In order to determine whether a certain function is completed conditionally, this routine can check the weighted adjacency matrix for the conditional code as filled in earlier and can access a system table of all such conditional system-provided functions SYSFCN. The system function itself is responsible for setting a flag at execution-time when the function is considered complete. This provides a general mechanism for implementing such a concept. The corresponding code generation routine is responsible for generating code to test for the completion flag of the function, and thereby flagging the completion of the assertion, which in turn triggers the execution of its successors. (See Section 4.5.1.5 for further explanation of the code that enables this mechanism).

In cases where the assertion involves a conditionally completed function, not only is the above flowchart table entry created, but also the node number of this assertion is added to a "conditional assertions list". Such a list is referenced by the "check conditions" subroutine (CHECOND), which is called by the Generate Flowchart control module (GENFLT already described), prior to generation of each subsequent flowchart entry. The conditional assertions list is checked in order to generate conditional control code to circumvent any operations that depend upon the completion of the assertion. The succeeding nodes which may depend on the completion of the conditional flowchart table entry are all nodes j such that

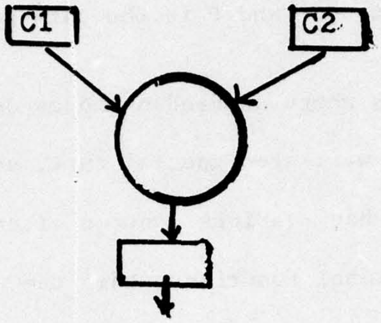
Pij=1

where i is the current node and P is the path matrix.

However, problems where succeeding nodes depend on more than one conditional function will need special care, and there are two cases. If a later assertion has various source items each of which is dependent on conditional functions, then the condition for execution of the assertion is the conjunction of completion of each of those conditional functions as illustrated in the following diagram:



where C1 and C2 are assertions using conditionally completed functions. If, on the other hand, a later assertion has a data item as source which depends on more than one conditional function, this implies that the functions are mutually exclusive. In such cases the condition for execution of the assertion is the disjunction (OR) of the completion of any of the precedent functions, as illustrated below:



where C1 and C2 are assertions using conditionally completed functions.

In order to generate the conditional control code before each descendant node, a flowchart table entry of the following form is created by the "check condition" (CHECOND) routine. Such an entry is generated before each node that is a descendant of the conditional function placed in the conditional assertions list:

NODE#	COND	ASSN NAME

Such flowchart table entries are used to generate the PL/1 conditional transfers (IF statements) later. In cases of dependencies on multiple conditional functions, adjacent "IF's" imply conjunction of the tests. For cases where a disjunction of the conditions is needed, described above, the conditional tests need to be separated by "or"s.

Returning to the flowchart table entry for assertions, RPLAB (replace label) is used only if the assertion uses the system-provided REPLACE function. It is the label to which the program branches after a replacement. It is determined by looking for the assertion target of the REPLACE in the dictionary and generating a label corresponding to that node number. The REPLACE feature is thus implemented as a special type of iterative loop whose starting point is the object of replacement and whose ending point is the assertion starting the replacement. It is in this way analogous to iterative loops implied by the FOREACH option described in Section 4.4.2. It would have been desirable, although not indicated here, to remove from this loop all nodes that do not depend on the starting node by moving them to the point preceding the "REPLACE" loop in a manner that was done for iterative loops.

One final check made by the IDASSN algorithm is to test whether or not the assertion is a "SUBSET" type assertion for a source file (Step 7); that is, whether the assertion is describing a condition under which a record is to be considered for the module. If it is this type of assertion (target is of the form "SUBSET .filename"), then code needs to be generated to branch to read the next record if the subset condition is not met. This is indicated in the flowchart table here as a COND type flowchart table entry (like the one depicted above) and a GOTO type entry (like the one described in Section 4.4.3.9). These flowchart entries produced here represent a conditional branch, and will correspond to the code: "IF ~SUBSET .filename THEN GO TO r", where r is the label of the READ for the next record.

The code generation routine that corresponds to the above assertion-description entry of the flowchart table will have the task of transforming such entries into PL/1 procedures. That code-generation procedure "Generate Procedure Code" (Section 4.5.1.4) will have to generate the procedure itself and its invocation, and any necessary control code, such as for replacement and stacking when necessary. Iterative control code is also generated when necessary for repeating data items (see Sections 4.4.3.7 and 4.5.1.6).

An example of an assertion entry in the flowchart table from the DEPSALE problem is below, and corresponds to the CALCCHRG assertion:

Node#	NodeType	Assn Name	CondFcn	RepLab	Text
4	ASSN	CALCCHRG	--	--	...

This indicates the assertion entry for CALCCHRG, which does not use any conditional functions nor replacement.

An example from the DEPSALE problem showing a replace label is the following:

Node#	NodeType	Assn Name	CondFcn	RepLab	Text
126	ASSN	TRYREPL	--	$L082	...

An example showing a conditional function is given in the corresponding code generation section (4.5.1.5).

4.4.3.6 Identifying Field Associations

The routine "Identify Field Associations" (IDFLDAS) is invoked by the "Generate Flowchart Table" routine whenever the current node is a field to check the weighted adjacency matrix for possible implicit assignments.

Algorithm IDFLDAS shows the Identifying Field Associations procedure and its generation of the flowchart table entry described below. The algorithm checks the column of the weighted adjacency matrix corresponding to the field in question in order to check the type of predecessor which the field has (Steps 2-5).

If a field is a member of a source record, no further code will have to be generated, because the field has a record or group predecessor and a value from the associated input command. Therefore, only a dummy flowchart table entry (of type FLDS) is created here with only its name for documentation purposes. A flowchart entry is created in any case so that all nodes have a corresponding entry in the

Algorithm IDFLDAS: Identifying Field Associations

Step 1. Fill flowchart table entry with node number, j.

Step 2. (check column of weighted adjacency matrix for source of field or interim; steps 2 through 5):

Set i=1.

Step 3. If Mij>0 then go to Step 6.

Step 4. Set i=i+1.

Step 5. Go to step 3.

Step 6. Set Node Type (in flowchart table entry) to a code depending on the predecessor as follows:

Case 1: If type is Field & Mij=1 (source record), then Set Node Type= FLDS .

Case 2: If type is Field & Mij=3 or 7 (explicit value), then Set Node Type= FLDP .

Case 3: If type is Field & Mij=4 (implicit value), then Set Node Type= FLDI .

Case 4: If type is Interim & Mij=3 or 7 (explicit value), then Set Node Type= INTP .

Case 5: If type is Interim & Mij=4 (implicit value), then Set Node Type= INTI .

Step 7. For Cases 3 and 5 above, enter the implicit source field in the flowchart table entry.

Step 8. Check attributes of assigned fields for compatibility.

Step 9. Return.

flowchart table.

If the field is a member of a target record, then it may also already have a value, by virtue of its being the target of an assertion which has that field as a successor and generates that value; $(\exists i)(Mij=3)$ where j is the node number of the field. In such a case, too, no further code need be generated, and therefore only a dummy flowchart table entry is created (of type FLDP) for documentation. This routine can easily check whether the field has an explicit value by virtue of being the target of an assertion, by checking the corresponding column within the weighted adjacency matrix (Step 2-6).

If, on the other hand, the adjacency matrix indicates that the field has an implicit source, [$(\exists i)(Mij=4)$ where j is the node number of the field], then a flowchart table entry (of type FLDI) needs to be created and will later be used to generate an assignment of the value to the field (Step 6, Cases 3 and 5; Step 7). Recall that the implicit source of the field was determined by the "Find Implicit Source" routine (Section 4.3.2.3.4) in the absence of explicit assertions. This includes correspondence of fields by such information as identical names and by keywords such as OLD and NEW. In such cases, a flowchart table entry of the following form is created:

NODE#	NODETYPE	TARGET FIELD	IMPLICIT SRC FLD

where

NODE# is the number of the node of this field as given by the order vector;

NODE TYPE is set to 'FLDI' to indicate that a field assignment code will need to be generated;

TARGET FIELD is the name of the field being assigned a value;

IMPLICIT Source Field is the name of the implicit source to this field.

The corresponding code generation routine will be able to generate the corresponding PL/1 assignment statement of the form:

target field = implicit source field;

An example of a flowchart table entry for an implicit assignment from the DEPSALE problem is the following:

NODE#	NODETYPE	TARGET FIELD	IMPLICIT SRC FLD
32	FLDI	JOURN.CUST#	TRANS.CUST#

4.4.3.7 Checking for Iterations

As explained in Section 4.4.2 on scope and iteration analysis and optimization, a table (DOTAB) of beginning and ending statements to be included in each iteration is created. Upon each loop through the order vector a routine is called (CHECKDO) to check the iteration table to see whether this is the starting statement of an iteration, and if so, a "DO-type" flowchart table entry is created with the following format:

Node#	Type	FOREACH name	Upper Type	Upper#	Upper Name

n	DO	Iteration Variable	F=fixed # times
			V=varying # times
				if "F", # times	if "V", EXIST name contains # of times

From this flowchart table entry, a PL/1 DO iterative statement is generated later by the corresponding code generation routine. Algorithm CHECKDO shows the process that generates the table entry. Likewise, the ending statement of an iteration is checked by the CHECEND routine, and if the most recent statement is the last one of an iteration, then an "END-type" flowchart table entry is generated (by Algorithm CHECEND) with the following format:

Algorithm CHECKDO: Check "Do-loops"

(Input data structure is DOTAB already described in Section 4.4.2; output data structure is the flowchart table entry described in this section).

Step 1. If there are no more entries in DOTAB table, then return.

Step 2. If the node number indicated in DOTAB not = current node number, then return.

Step 3. Create flowchart table entry for iteration (depicted in text).

Step 4. Fill entry with (a) current node number, (b) "DO" type, and (c) iteration variable (from DOTAB).

Step 5. Retrieve type of iteration (fixed or variable number of times) and fill in F or V respectively in entry as depicted.

Step 6. Fill in number of repetitions (if fixed) or EXIST name (if variable).

Step 7. Examine next entry in DOTAB; go to Step 2 (possible beginning of other iterations).

Algorithm CHECEND: Check for Generation of "END"

Step 1. If there are no more entries in DOTAB then return;

Step 2. If the node number indicated in DOTAB not = current node number, then return.

Step 3. Generate "END" flowchart table entry as depicted.

Step 4. Examine next entry in DOTAB then go to Step 2 (possible end of other iterations).

Node#	Node Type	
n	END	

to represent the end of the iteration.

An example of such entries for an iteration from the DEPSALE *problem are the following:*

Node#	Type	FOREACH name	Upper Type	Upper#	Upper Name
112	DO	FOREACH-TRITEM	V	--	EXIST.TRITEM

Node#	Node Type	
21	END	

4.4.3.8 Checking for Label Generation

As noted earlier, for example in the case of replacement, statement labels which are referenced are generated occasionally by putting them in a "label table". As the GENFLT control routine loops through the order vector, it invokes the "Check label" routine (CHECLAB) to check the label table to see whether any label needs to be generated for the current statement.

If so, a flowchart table entry with the following format is generated by the CHECLAB Algorithm:

Node#	Node Type	Label Name
	LAB	x

This is used by the corresponding code generation routine to generate a PL/1 statement label.

An example from the DEPSALE problem is for the label referenced for replacement:

Algorithm CHECLAB: Check for Labels

(Input data structure: "label table", the table of labels to be generated)

Step 1. Iterate through "label table", performing Step 2 for each entry; return.

Step 2. If current node=node to be labeled, then create flowchart table entry for label as depicted.

Node#	Node Type	Label Name
82	LAB	$L082

The label name is formed by a '$L' concatenated with the node number (82 in this example) of the node receiving the label.

4.4.3.9 Identifying Housekeeping and End-of-Program Tasks

At the end of generation of flowchart entries for all the nodes of the order vector, there still remains some "housekeeping" code to be generated. Control code is needed to branch to process the next set of records. This is accomplished by generating a "GOTO-type" flowchart entry:

Node#	Node Type	Name
	GOTO	x

where x is the label (saved by IDIOCD) which starts the program.

An example from the DEPSALE problem which branches back to read the next transaction is the following:

Node#	Node Type	Label Name
	GOTO	$RD-TRANS

Before branching back, however, several events are required. Specifically, all the CHOICE variables, conditional flags, and replacement flags must be reset so that they can be set again in the next cycle. This is accomplished by generating "reset-type" flowchart entries of the form:

Node#	Node Type	Name
	RSET	x

where x is the flag or switch being reset. Note that variables associated with functions (such as total variables) are not reset automatically here, but rather by the function itself upon completion.

An example of a "reset" flowchart table entry from the DEPSALE problem is the following:

Node#	Node Type	Name
	RSET	CHOICE.SALE

which resets the CHOICE name SALE.

An "end-type" flowchart entry is also generated to mark the end of the program.

Algorithms IDGOTO and IDRSET consist simply of code to generate these two kinds of entries respectively, and are therefore omitted.

4.4.3.10 The Flowchart Table Report

The entries of the flowchart table are formatted and printed out by this routine, "Generate Flowchart Report" (GFLTRPT), so that a system programmer or interested user of MODEL could check the flow of the generated program. Each entry of the flowchart table is accepted by this routine as input. After branching on the flowchart entry type, it produces a report with a line corresponding to each entry as output. A schema of each kind of line of the report is given in Figure 4.16a and a sample flowchart report appears in Figure 4.16b. Each entry contains the following:

the NODE NUMBER, which is the same as in the flowchart table;

the NAME of the item at that node, if any;

a DESCRIPTION of the node; and

the EVENT to be performed (an English summary of the PL/1 statements at that node).

4.4.3.11 Generating Table of Data Structures Declarations

Just as a language-independent table is generated to represent a flowchart of the executable portion of the object program, this routine generates a table of all the variables and attributes for which PL/1 declarations will have to be generated. Generating the necessary declarations, then, is a two-step process, as shown in Figure 4.17. This first routine, "Generate Declarations Table" (GDCLTAB) accepts, as input, the stored MODEL data descriptions, and

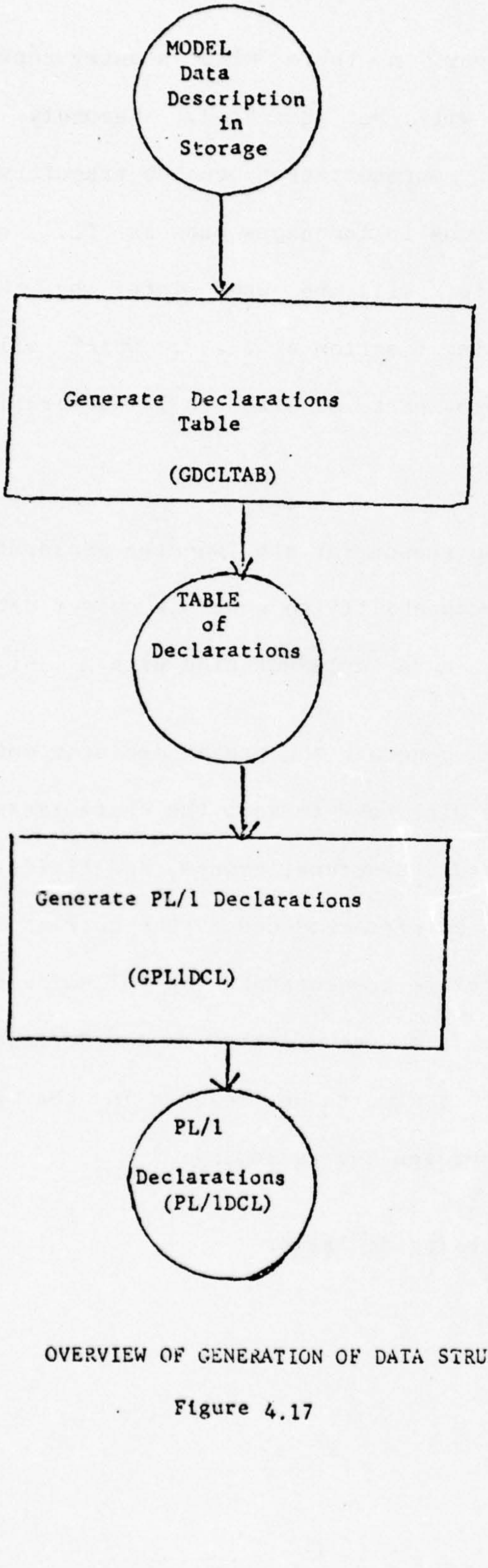
Node#	Node Type	Description	Event
	module(MODL)	Module Name	Proc Heading
	file	FILE	
	record(RECD or RPTN)	RECORD	READ, WRITE, or REWRITE command (with approp. Parameters)
	group (GRP)	GROUP	
	field(FLD)	FIELD:	
		target of assert. A	--
		implicit assign.	x=y
		in source record x	--
	interim (INTR)	INTERIM (same as for field)	(same)
	assertion (assn)	ASSERTION	CALL assertion (if replacement then indicates "repl to x")
	storage(CARD DISK,TAPE)		STORAGE DEVICE
	GOTO		GO TO --
	DO	ITERATION	ITERATE FOR EACH X
	END	END OF ITERATION or END OF PROGRAM	
	LAB	(LABEL):	(X):
	RSET	RESET FLAGS	r=0

Figure 4.16a

Flowchart Table Report Entry Types

```
                                      FLOWCHART REPORT
NODE#       NAME                          DESCRIPTION                      EVENT

   3 MINSALF                         MODULE NAME                        PROCEDURE HEADING
   9 OLD.INVEN                       FILE NAME
  21 SALETRAN                        FILE NAME
  15 SALEDECK                        STORAGE DEVICE
  16 SALEREC                         RECORD                             'READ SALEREC
  22 SALETRAN.CUST#                  FIELD IN RECORD SALEREC
  23 SALETRAN.QUANTITY               FIELD IN RECORD SALEREC
  24 SALETRAN.STOCK#                 FIELD IN RECORD SALEREC
  19 SALESLIP.CUST#                  FIELD(IMPLICIT ASSIGN)             SALESLIP.CUST#=SALETRAN.CUST#
  20 SALESLIP.STOCK#                 FIELD(IMPLICIT ASSIGN)             SALESLIP.STOCK#=SALETRAN.STOCK#
  26 TRINV                           ASSERTION                          CALL TRINV
  14 POINTER.OLD.INVREC              TARGET OF SPECIAL ASSERTION
  13 OLD.INVREC                      RECORD                             READ OLD.INVREC
  10 OLD.INVEN.QOH                   FIELD IN RECORD OLD.INVREC
  11 OLD.INVEN.SALPRICE              FIELD IN RECORD OLD.INVREC
  12 OLD.INVEN.STOCK#                FIELD IN RECORD OLD.INVREC
   1 CALCCHRG                        ASSERTION                          CALL CALCCHRG
   6 NEW.INVEN.SALPRICE              FIELD(IMPLICIT ASSIGN)             NEW.INVEN.SALPRICE=OLD.INVEN.SALPRI(
   7 NEW.INVEN.STOCK#                FIELD(IMPLICIT ASSIGN)             NEW.INVEN.STOCK#=OLD.INVEN.STOCK#
  27 UPDQUAN                         ASSERTION                          CALL UPDQUAN
   5 NEW.INVEN.QOH                   FLD:TARGET OF ASSERTION UPDQUAN
  18 SALESLIP.CHARGE                 FLD:TARGET OF ASSERTION CALCCHRG
   8 NEW.INVREC                      RECORD                             REWRITE NEW.INVREC
  25 SLIPREC                         RECORD                             WRITE SLIPREC
   4 NEW.INVEN                       FILE NAME
  17 SALESLIP                        REPORT NAME
   2 INVDISK                         STORAGE DEVICE
                                                                        GO TO $RD_SALETRANS
                                     (LABEL)                            ($FINISH):
                                     END OF PROGRAM
```

Figure 16b Sample Flowchart Report



OVERVIEW OF GENERATION OF DATA STRUCTURES DECLARATIONS

Figure 4.17

produces, as output, a table with an entry representing each name declaration that will be generated. Secondly, this language-independent table representation can be transformed later into data structure declarations in languages such as PL/1 or COBOL. In our case, this table will be used later by the "Generate PL/1 Declarations" routine (Section 4.5.1.9), which will transform each table entry into part of the PL/1 hierarchic data structure declarations.

As before, the reason for the two-step declaration generation is modularity and the capability to generate object data structures other than PL/1 at some future implementation with a minimum of change.

In order to generate the proper declarations, the declarations generating routine will have to know the characteristics of each file, its component records structure, groups, and fields. The format of the Declarations Table entries produced by the current routine (Generate Declarations Table) are presented in the following chart in Table 4.8. Each schema here is a type of entry in the Declarations Table which, in turn, represents a name to be declared in the generated program. The table entry contains the following:

(1) the name being declared;

Name	Type	Level	Addition Declarative Information				
$x\left\{\begin{matrix}S\\T\\u\end{matrix}\right\}$	'FL' (File)	0	(I/O mode) I (input) 0 (output) U (update)		Organization S (sequential) I (indexed)		
x	'FR' (File in structure)	first					
x-S	'RC' (record)	0	Length Type F (fixed) V (variable)		Length m		
x	'RR' (record structure name)	second					
x	'GP' (group)	n	repetitions				
x	'FD' (field)	n	repetition	Field Type B (binary) C (character) D (fixed)	Field-Length-Type F (fixed) V (variable)	min-length p	max-length q

Table 4. 8 Data Declarations Table

(2) the type of name;

(3) the "level" within the object hierarchic data structure (with the file name at the top, record name second, and groups and fields below that); and

(4) other information necessary for the declaration, as indicated in Table 4.8.

A FILE declaration gets two entries in the table. The first (described in the table as type 'FL') has the indicated suffix (described in Section 4.4.3.3) added to the file name as given by the user. This is the name that will be used to declare the external file for later use by the PL/1 compiler and the operating system, and the file name referenced by the generated input/output statements. The entry conveys a unique name for the file, the file input/output mode, and the file organization. The suffix is added in the first entry to distinguish it from the second type of entry for files (described in the table as type 'FR') which has the file name as given by the user. The second file name is used for the declaration of the highest level in the PL/1 hierarchic data structure. The reason for the two names (already explained in Section 4.4.3.3) is that the user can qualify field names by using the name of the file containing them. For example, if a file named INVEN has a field named STOCK# within it, then making the original file name the name of the hierarchic tree-structured buffer allows the field to be referred to by the user as INVEN.STOCK# (to distinguish it from the same-named field in other files).

The RECORD declaration also gets two entries in the table. The first (denoted in the table as type 'RC') contains information on the length and type of record for the declaration of a buffer string within the area of the generated PL/1 program, which is used by the I/O operations. The name of this buffer is formed as the name of the record with '-s' concatenated to distinguish it from the record name as given by the user. The second entry for a record has the name as given by the user. This name will be used as the second-highest level of the PL/1 hierarchic data structure (after the file name). This allows the user to qualify a field name alternatively by the name of the record it is in.

The GROUP declaration entry needs the name and level of the group and the number of repetitions if any.

The FIELD declaration entry contains all the information that will be needed to declare the field at the indicated level of the hierarchic structure. This includes the length and field type attributes indicated in the chart.

These table entries are created by the "Generate Declarations Table" algorithm (GDCLT). It retrieves each file description (Steps 1-3), and for each file's record description, the flowchart table entry for the record string buffer is generated (Subroutine FREC). The table entries for the record component groups and fields are generated by recursively "climbing" down the tree-structure of the data structure implicit in the stored MODEL data description statements (Subroutine FORM-TREE). Furthermore, this routine links together the Declaration

Algorithm GDCLT: "Generate Declaration Table"

Variable names used in this algorithm:

Declaration Table: depicted in Table 4.8

SOURCE-ARRAY: array to hold names of files that are source only.

TARGET-ARRAY: array to hold names of files that are target only.

SOURCE-TARGET-ARRAY: array to hold names of files that are both source and target

S: stores name of each source-only file
T: stores name of each target-only file
ST: stores name of each file that is both source and target

```
Step 1. Retrieve all source-only file names and put into SOURCE-ARRAY

Step 2. Retrieve all target-only file names and put into TARGET-ARRAY

Step 3. Retrieve all source-and-target file names and put into SOURCE-TARGET-ARRAY

Step 4. For each source-only file name in SOURCE-ARRAY (call it S):

   Generate both file entries (as depicted in Table 4.8).
   Call FREC(1)
   Call FORM-TREE(S,1)

Step 5. For each target-only file name in TARGET-ARRAY (call it T):

   Generate both file entries (as depicted in Table 4.8).
   Call FREC(2)
   Call FORM-TREE(T,1)

Step 6. For each source-and-target file name in SOURCE-TARGET-ARRAY (call it ST):

   Generate both file entries (as depicted in Table 4.8).
   Call FREC(1)
   Call FREC(2)
   Call FORM-TREE(ST,2)

Step 7. Return.
```

Algorithm GDCLT : Subroutine FREC(I-O-CODE)

(forms declarations table entries for records string buffer)

Parameter I-O-CODE: indicates whether file is source or target (1=input, 2=output)

Step 1. Create table entry with name and type as depicted in Table 4.8.

Step 2. Retrieve record name and if corresponding file is both Source and Target, modify the record string name with prefix 'OLD_' or 'NEW_'

Step 3. Retrieve record length and type and fill in table.

Step 4. Return.

Subroutine FORM-TREE(NAME,LEVEL)

(recursive routine to generate table entries representing hierarchical record structure):

parameters: name -- data name being declared

level -- level in the tree

Step 1. If name type is FLD then generate table entry for fields as depicted; return.

Step 2. Create table entry for current level in tree (file, record, or group as in Table 4.8)

Step 3. Retrieve all sub-members of current level name.

Step 4. For each sub-member (call it X) call FORM-TREE(X,LEVEL+1).

Table entries in the order in which the declarations are to be made.

4.5 Code Generation

This phase of the Processor proceeds after specification analysis, precedence determination, program design, and flowchart creation have been completed. Recall that had there been user errors during syntax analysis or specification analysis, then neither the flowchart creation nor the code-generation phases would be reached. As seen in Figure 4.18, the code generation phase accepts as input the flowchart table and the declarations table produced in the previous phase, and produces as output a complete PL/1 program ready for compilation.

4.5.1 Generation of PL/1 Program

The control program for generating the complete PL/1 program (CODEGEN), as shown in Figure 4.18, accepts the table of declarations and the flowchart table created during the previous phase as input. The various types of entries of the flowchart table were described fully in Section 4.4. They are the entries of type ASSN, RECD, RPTN, FLDI, INTI, GOTO, MODL, LAB, END, DO, RSET, and COND as described in Section 4.4. This phase produces, as output, the complete PL/ı program and a code-generation report. The files to which code is written are described below.

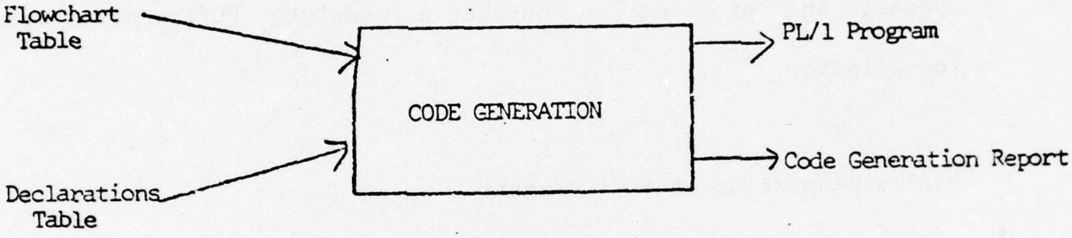


Figure 4.18

Overview of the Code Generation Phase

PL/1 was chosen as the object language because of its versatility, ease and richness in data structures, control structures, and other language features suitable to business data processing programs, and its growing acceptability in this realm. Nothing in the Processor up to this point, however, depends on the object language being PL/1, and this in fact was a major reason for the modularity. Therefore, generating a program in another object language, such as COBOL, would be a straight-forward though tedious conversion of the following code-generation procedures.

Generating the PL/1 program code, as can be seen in Figure 4.19, is accomplished by processing the input tables described above and invoking the appropriate code-generation sub-routine. Algorithm GENPL1 shows the Generate PL/1 Program Control procedure. The executable PL/1 code is generated by inputting the flowchart table entries one at a time, and invoking the code-generation routine that corresponds to the type of operation (Steps 2-3). The tests for each type of code to be generated are in decreasing order of frequency. These include code-generation routines for input-output operations, for invoking and writing of object sub-procedures, for assigning implicit values to fields, and for generating control structures.

The executable PL/1 code is written out to the "PL1EX" file, while associated PL/1 "ON" conditions are written to the "PL1ON" file. The PL/1 procedures (which contain asssertions plus functions) are written to the PL1PROC file. The PL/1 code for declaring the object data items is written to a "PL1DCL" file.

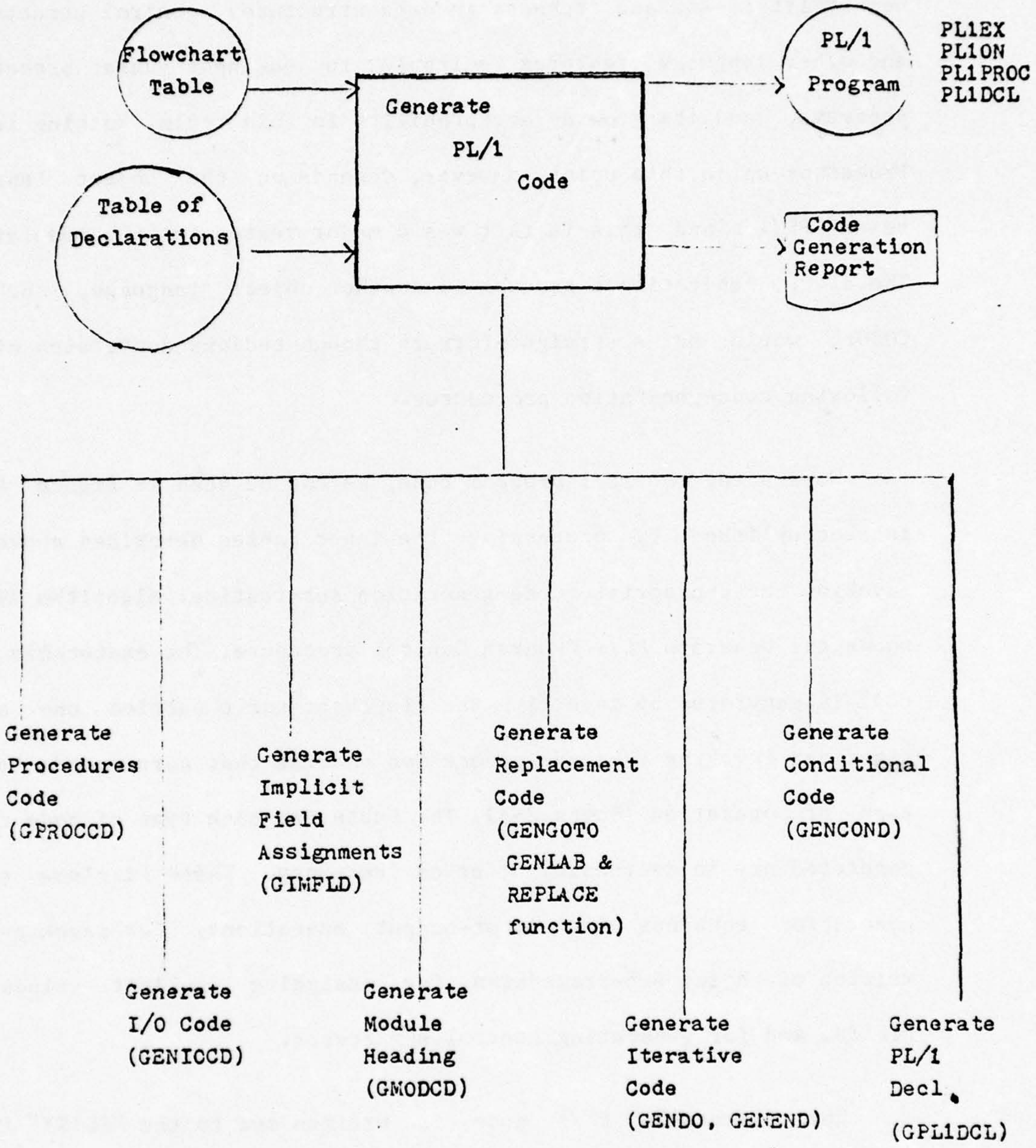

Flowchart Table
Table of Declarations
Generate PL/1 Code
PL/1 Program
PL1EX
PL1ON
PL1PROC
PL1DCL
Code Generation Report
Generate Procedures Code (GPROCCD)
Generate I/O Code (GENIOCD)
Generate Implicit Field Assignments (GIMFLD)
Generate Module Heading (GMODCD)
Generate Replacement Code (CENGOTO GENLAB & REPLACE function)
Generate Iterative Code (GENDO, GENEND)
Generate Conditional Code (GENCOND)
Generate PL/1 Decl (GPL1DCL)

Figure 4.19

Components of Generating PL/1 Code

Algorithm GENPL1: Generate PL/1 Program Control Procedure

[Subroutines called: GPROCCD, GENIOCD, GIMFLD, GENGOTO, GMODCD, GENLAB, GENEND, GENDO, GENRSET, GENCOND]

Step 1. Read next flowchart table entry; if end-of-file, then go to Step 4.

Step 2. Branch on flowchart entry type and call appropriate routine as follows:

If ASSN, then Call GPROCCD.

If RECD or RPTN, then call GENIOCD.

If FLDI or INTI, then call GIMFLD.

If GOTO, then call GENGOTO.

If MODL, then call GMODCD.

If LAB, then call GENLAB.

If END, then call GENEND.

If DO, then call GENDO.

If RSET, then call GENRSET.

If COND, then call GENCOND.

Step 3. Go to Step 1.

Step 4. Return.

All these code-generation sub-routines together with the necessary transformations are described in the following sub-sections. The algorithms that are used in the following sub-sections are explained by referring to the input flowchart table entry, the generated PL/1 code, and the transformations between them. Tables will be used to show the code that is generated in the various cases.

4.5.1.1 Generating the Procedure Heading Code

The routine for generating the heading of the PL/1 module for which code is being generated (GMODCD) is called by the Generate PL/1 control routine after the flowchart entry for a module statement is read. This routine has the simple task of accepting, as input, the flowchart table entry for the module name (as was described in Section 4.4.3.2), and producing, as output, the PL/1 procedure declaration:

```
+-------------------------------------------+
|                                           |
|Module name: PROC OPTIONS (MAIN);          |
|                                           |
+-------------------------------------------+
```

as the first statement of the program. This statement, unlike the executable statements produced in this section, is written to the PL1DCL file (rather than the PL1EX file) in order that it appear as the first statement in the program. (The PL1DCL file with the declarations yet to be written to it, will precede the PL1EX file with executable statements in the final program).

4.5.1.2 Generating Input/Output Code

The routine for generating input/output code (GENIOCD) is invoked by the generate PL/1 code control routine after reading a flowchart entry that corresponds to an I/O command. It accepts, as input, an entry from the flowchart table corresponding to a record (FLOWTAB_REC), which was created and described in Section 4.4.3.3. The entry in the table already has all the necessary information relevant to generating the appropriate PL/1 I/O statement, including the file organization, input/output direction or mode, the key field, etc. This routine generates the PL/1 READ, WRITE, or REWRITE Statements with the appropriate parameters based on the flowchart table entry, as well as any control code or condition code associated with the input/output operation.

To summarize the transformations of Algorithm GENIOCD from the flowchart representation of the input/output code to the corresponding PL/1 statements, Table 4.9 is given here instead of an algorithm form for the sake of clarity. If the value of the components of the flowchart table entry are as indicated in the first four columns of Table 4.9, then the code generated is indicated in the last two columns. The upper case letters represent part of the actual PL/1 string being generated, whereas the lower case letters are the meta-names of the items obtained from the flowchart table during program generation.

I/O MODE	ORG	KEYED	KEY NAME	GENERATED PL/1 I/O STMT WITH ASSOCIATED CONTROL CODE	OTHER PL1 CODE GENERATED
RD	S	no	--	READ FILE(filenameS) INTO(recname-S);	ON ENDFILE (filenameS) GOTO $FINISH;

RD	S	yes	k	$RD-filenameS:
				READ FILE (filenameS)
				INTO (recnameS);
				DO WHILE
				(k< POINTER. Recname);
				WRITE FILE (filenameT)
				FROM (recname-S);
				READ FILE (filenameS)
				INTO (recnameS);
				END;
				IF k> POINTER. Recname
				THEN CALL $NOTFOUND
				(filenameS);
					ON ENDFILE
					(filenameS)
					CALL $NOTFOUND
					('filename'S);

RD	I	yes	k	READ FILE (filenameS) (filenameS) INTO (recname-s) KEY(POINTER. Recname)	ON KEY CALL $NOTFOUND ('filenameS')
WR	S	yes or no	--	WRITE FILE(filenameT) FROM (recname-S);	

RW	I	yes	k	REWRITE FILE
				(filenameT)
				FROM (recname-S)
				KEY(POINTER. Recname);	

Table 4.9

Input/Output Transformations from Flowchart Table to PL/1

For example, the "READ" flowchart table record generated for SALEREC of the DEPSALE problem, was shown in Section 4.4.3.3. Such an entry describes a READ statement from a sequential non-keyed file, and this routine would generate the following PL/1 code:

```
READ FILE (TRANS) INTO (SALEREC_S);
```

Notice the use of the special POINTER names in the cases of keyed files. Their evaluation will have automatically been previously executed by virtue of their precedence; i.e. the POINTER name is precedent to the record name.

The second case in this chart is more complex because it involves searching the sequential file for a record whose key field is the desired one to which there is a pointer. Since the file is sequential, searching for the record involves successive READS in a loop. The WRITE statement in this loop is generated only in the case that the sequential file is to be updated (both input and output). There are other functionally equivalent ways to write PL/1 code, but these were chosen for ease of implementation, style, or efficiency.

4.5.1.3 Generating Code for Variable Length, Optional, and Variably Repeating Data

When all the fields of a record are of a fixed length and occur a fixed number of times the read (or write) statements that are generated as shown above are sufficient to cause the data in the record to be transferred to (or from) the PL/1 hierarchic data structure. However, the MODEL language as previously indicated, has facilities to describe variable-length fields or variably-existing groups or fields, where length, existence, or repetition is to be determined dynamically by evaluating user-provided assertions. The difference between such general facilities of MODEL for variability and the less general ones in PL/1 (the PL/1 REFER option and variable-length strings in PL/1) have already been described in Section 4.4.3.4. Therefore, generation of input/output code for records with such items cannot simply transfer the data directly in and out of the PL/1 data structure. Instead, the PL/1 code to be generated here must simulate the variability provided in MODEL. As explained in the corresponding section within flowchart generation in Section 4.4.3.4, the information necessary for such data "packing" and "unpacking" operations has *already been identified* in the flowchart entry (i.e. the group and fields that have such variable length or repetition are indicated).

Algorithms "Generate Packing" and "Generate Unpacking" show the generation of code for variable length, optional, and variably repeating data by generation of "unpacking′ operations following the READ and generating "packing" operations before the WRITE. The data structures and strategy that are used in the generated program are as follows: for each file there are two buffers -- one which is just a PL/1 string for each record of the file and the other buffer is the PL/1 hierarchic data structure. The process of "unpacking" is the scanning of the string buffer and copying of the exact data amount to the hierarchic data structure, while the process "packing" is in the reverse direction.

Algorithm Generate Unpacking
(generation of code for input variable-length, optional, and variably-repeating data by "unpacking")

Names and Data Structures used by this algorithm:
Flowchart Table Entry for variable length records -- already described in Section 4.4.3.4. Its components are NAME, TYPE, #SUBSCRIPTS, SUB1, SUB2, EXIST-PROC, ARITY, FIELD-TYPE, FIELD-LEN-TYPE, MIN-LENGTH, MAX-LENGTH, LEN-PROC.

RECSTRING: string buffer for record

RECNAME.NAME: a field or group name in hierarchic data structure of record called recname.

NSTR: level number in hierarchical tree.

SUBSCRIPT-STACK: Stack of PL/1 index names to repeating fields and groups; has the form "Inn" where "nn" is the level in the tree.

Sub-structure: name used to refer to a sub-tree in the hierarchical data structure.

Step 1. Initialize subscript stack.

Step 2. Set NSTR=0 (substructure number).

Step 3. Generate 'I=1' to initialize buffer pointer in object program.

Step 4. Call UNPACK(1) for each top-level member of the record (parameter is next index to use as subscript or do-variable).

Step 5. Return.

Algorithm Generate Packing
(generation of code for output variable-length, optional, and variably-repeating data by "packing").

Step 1. Initialize subscript stack.

Step 2. Set NSTR=0 (substructure number).

Step 3. Generate "record-string = ''" in order to initialize output buffer.

Step 4. Call PACK(1) for each top-level member of the record (parameter is next index to use as subscript or do-variable).
Step 5. Return.

PL/1 code is generated here for dynamic computation of length, existence, or repetition. After a READ statement is generated for a record with any of the above variability, code is generated to "unpack" the data by calling "Generate Unpacking" and its subroutines. Code is generated to scan the input buffer from left to right on a field-by-field basis. If a field indicated in the flowchart table entry has variable length, a CALL is generated to the assertion which determines its length; i.e. the assertion whose target is LEN.X, where X is the name of the field. Thus, the length of the field is known, and further code is generated to transfer the data in the buffer to the corresponding field name in the PL/1 hierarchic data structure. Likewise, if a field or group is indicated to repeat a variable number of times, code is generated to CALL the assertion that determines the number of repetitions; i.e. to the assertion whose target is EXIST.X where X is the name of the variably-repeating group or field. A "subscript stack" (a stack of the form I01, I02, etc.) is maintained in case repeating groups or fields are nested. These indices are used for subscripting and for PL/1 "DO-loop" variables. Further code is generated to move the exact number of repetitions of the item in the buffer to the corresponding name in the PL/1 hierarchic data structure. In all these cases, the PL/1 data names allow for the maximum amount of length or repetition.

Subroutine: (UN)PACK (next-index)
(a recursive procedure which supervises generation of code for (un)packing records before they are read/written).
Parameter "next-index" is next index to use in a do-loop or for subscripting.

```
Step 1. Set NSTR=NSTR+1 (next sub-structure).

Step 2. If member type is 'F' (field), then go to Step 3.
If member type is 'G' (group), then go to Step 9.

Step 3. (UN)PACKFLD -- Steps 3 through 7:

   Branch on case:

Step 4. Case 1: #SUBSCRIPTS=0 (field occurs only once)
   If packing, then Call GEN-MOVE-INSTR-FOR-PACKING ;
   if unpacking,then Call GEN-MOVE-INSTR-FOR-UNPACKING.
   Go to Step 8.

Step 5. Case 2: #SUBSCRIPTS=1 (field occurs a fixed number of times)
   Push subscript on to stack.
   Generate 'DO Inn=1 TO subscript1'.
   if packing, then Call GEN-MOVE-INSTR-FOR-PACKING ;
   if unpacking, then Call GEN-MOVE-INSTR-FOR-UNPACKING.
   Generate 'END' (end of loop).
   Pop subscript from stack.
   Go to Step 8.

Step 6. Case 3: #SUBSCRIPTS=2 & SUBSCRIPT2=1 (optional field):
   Generate 'CALL exist-procedure'.
   Generate 'DO Inn = 1 TO EXIST. Name'.
   if packing, then Call GEN-MOVE-INSTR-FOR-PACKING;
   if unpacking, then Call GEN-MOVE-INSTR-FOR-UNPACKING.
   Generate 'END'
   Go to Step 8.

Step 7. Case 4: #SUBSCRIPTS=2 & SUBSCRIPT2>1 (field repeats a variable
number of times)

   Push subscript on to stack.

   Generate 'CALL exist-procedure'.
   Generate 'DO Inn = 1 TO EXIST. Name'.
   if packing, then Call GEN-MOVE-INSTR-FOR-PACKING;
   if unpacking, then Call GEN-MOVE-INSTR-FOR-UNPACKING.
   Generate 'END'
   Pop subscript from stack.

Step 8. Return.
```

Subroutine (UN)PACK (continued)

```
Step 9. (UN)PACKGRP steps 9 through 13:
     Branch on case:

Step 10. Case 1: #SUBSCRIPTS=0 (group occurs only once)

   Call (UN)PACK (next_index) recursively for each member of the group.
   Return.

Step 11. Case 2: #SUBSCRIPTS=1 (group repeats a fixed number of times)

   Push subscript on to stack.
   Generate 'DO Inn=1 TO subscript1'.
   Call (UN)PACK (next_index + 1) recursively for each member of the group.
   Generate 'END'
   Pop subscript from stack.
   Return.

Case 3: #SUBSCRIPTS=2 & SUBSCRIPT2=1 (optional group):
   Generate 'CALL exist_procedure'.
   Generate 'DO Inn = 1 TO EXIST. Name'
   Call (UN)PACK (next_index+1) recursively for each member of the group.
   Generate 'END'
   Return.

Case 4. #SUBSCRIPTS=2 & SUBSCRIPT2>1 (group repeats a variable number of times)

   Push subscript on to stack.
   Generate 'CALL exist_procedure'.
   Generate 'DO Inn = 1 TO EXIST. Name'.
   Call (UN)PACK (next_index + 1) recursively for each member of this group.
   Generate 'END'
   Pop a subscript from the stack.
   Return.
```

Optionality is effected whenever the item is defined to repeat zero or more times. The above procedure handles optionality as well because the EXIST variable would return zero, and no repetitions for the item would be moved or used.

Similar code is generated when the variable items are in an output record, except that the code is generated to "pack" the data from the PL/1 data names to the output buffer. The "Generate Packing" procedure of the above Algorithm is called before each WRITE of a record with variable repetition, existence, or length. For variable-length fields, code is generated to call the length-evaluating procedure and then to move the exact length of the field to the output buffer. Similarly, for variable-repetition, code is generated to CALL the procedure determining the number of repetitions and then to move the exact number to the output buffer.

In all these algorithms, generated code is between the quotes. Upper case within quotes represents object names in the generated code, whereas lower case between the quotes represents names to be generated by the Processor. Upper case elsewhere represents meta-names or procedures in the Processor.

The code that is generated here is made clearer by the following example. Consider the following sample MODEL statements for a record R of a source sequential file F:

Subroutine GEN-MOVE-INSTR-FOR-UNPACKING

```
Step 1. If FIELD-TYPE='C' or 'N' (character or numeric character) and
if FIELD_LEN_TYPE=F (fixed length) then go to Step 5.

Step 2. If FIELD_TYPE='C' or 'N' and FIELD_LEN_TYPE='V' (variable
length) then go to Step 6.

Step 3. If FIELD_TYPE='B' (binary) then go to Step 7.

Step 4. If FIELD_TYPE='F' (fixed decimal) then go to Step 8.

Step 5. (character or numeric fixed-length field):
   Generate 'recname.name(...)= SUBSTR (recstring,I,min-length)'
   Generate 'I=I+min-length'
   Return.

Step 6. (variable-length character or numeric field):
   Generate 'CALL len-procedure'
   Generate 'recname.name(...)= SUBSTR
   (recstring,I,LEN.name)'
   Generate 'I=I+LEN.name'
   Return.

Step 7. (field is binary) If min-length (number of bits) < 16 then
#bytes=2; else #bytes=4; go to Step 9.

Step 8. (fixed decimal) Set #bytes=ceil(.5*(min-length+1))

Step 9. Generate 'UNSPEC (recname.name(...))= UNSPEC(SUBSTR
(recstring, I, #bytes))
Generate 'I=I+#bytes'

Step 11. Return.
```

Subroutine GEN-MOVE-INSTR-FOR-PACKING

```
Step 1. If field type is 'C' or 'N' then go to Step 3.

Step 2. If field type is 'B' or 'F' then go to Step 5.

Step 3. Generate 'recstring=recstring||recname.name(...)'

Step 4. Return.

Step 5. Generate 'UNSPEC (recstring)= UNSPEC (recstring)|| UNSPEC
(recname.name(...))

Step 6. Return.
```

(note: in all above cases, '...' is the nested subscripts from the subscript stack)

```
R IS RECORD(A,B,C(1:10));
A IS FIELD(CHAR(5));
B IS FIELD(CHAR(1:20));
C IS GROUP(C1,C2);
C1 IS FIELD(CHAR(2));
C2 IS FIELD(CHAR(3));
...
C_ASSN: ... EXIST.C = ...
B_ASSN: ... LEN.B = ...
```

Then the following code is generated here:

```
READ FILE(F) INTO(R_S);
R.A=SUBSTR(R_S,I,5);
I=I+5;
CALL B_ASSN;
R.B=SUBSTR(R_S,I,LEN.B);
I=I+LEN.B;
CALL C_ASSN;
DO I01=1 TO EXIST.C;
   R.C1=SUBSTR(R_S,I,2);
   I=I+2;
   R.C2=SUBSTR(R_S,I,3);
   I=I+3;
END;
```

4.5.1.4 Generating Procedures Code

The "Generate Procedures Code" (GPROCCD) routine is invoked by the "Generate PL/1 Code" control routine after reading a flowchart table entry that corresponds to an assertion. As input it accepts an entry from the flowchart table corresponding to an assertion, whose format was described in the "Identify Assertions Information" (Section 4.4.3.5). As output it produces the PL/1 code which embodies the assertion, a PL/1 CALL to the procedure, and any necessary control code to govern the execution of the procedure invocation.

Algorithm (GPROCCD) generates the code presented below. It also generates the conditional code and replacement code described in the next few sub-sections. The user-provided assertions are implemented as PL/1 procedures in order to achieve a top-down and modular structure. For each assertion flowchart table entry, this routine generates the following PL/1 procedure code which it outputs to a PL/1 procedures file (PL1PROC):

```
+--------------------------------------------+
|                                            |
|Assertion: PROC;                            |
|                                            |
| assertion text                             |
|                                            |
| RETURN;                                    |
|                                            |
| END;                                       |
|                                            |
+--------------------------------------------+
```

where "assertion" is the name of the assertion as originally provided by the user, and "assertion text" is the text of the procedure from the flowchart table entry. Furthermore, in order to invoke the

Algorithm GPROCCD: Generated Procedure Code

(Input to this algorithm is the flowchart table entry for assertions (with components ASSN-NAME, FCN, RPLAB, TEXT) that was created and described in Section 4.4.3.5)

Step 1. Generate code to call procedure: "CALL Assn-name" (sent to file PL1EX)

Step 2. If RPLAB not=' ' (assertion uses REPLACE function) then generate code for replacement (see box of code in Section 4.5.1.7, sent to file PL1EX)

Step 3. If FCN not=' ' (assertion uses a conditional function) then generate code for testing the function's completion (see first box of code in Section 4.5.1.5, sent to file PL1EX)

Step 4. Generate body of assertion (see first box of code in Section 4.5.1.4, sent to PL1PROC file)

Step 5. Return

generated sub-procedure, the following code is generated in the PLIEX output file which contains the main procedure of the executable generated PL/1 statement:

```
+--------------------------------------------+
|                                            |
|CALL assertion;                             |
|                                            |
+--------------------------------------------+
```

where "assertion" is the name of the generated procedure as was given by the user.

In addition to the generation of the procedure and its invocation, there is a possibility that PL/1 control code is necessary in cases of conditionality, repetition, and replacements, subjects which are covered below.

4.5.1.5 Generating Conditional Code

If the current flowchart table entry (of type "assn") has the conditional flag up, it indicates that the assertion involves the use of a system-provided function, whose completion is conditional, as explained in Section 4.4.3.5. In such a case, code has to be generated here to flag the completion of the node when the function finishes, which in turn triggers execution of any operations that depend on the completion of the function. This code is generated as part of the GPROCCD algorithm which already has been shown in Section 4.5.1.4. In these cases, conditional PL/1 code has to be generated in the form:

```
+-------------------------------------------+
|                                           |
|IF function-name_COMPLETED THEN DO;        |
|                                           |
|function-name_COMPLETED='0'b;              |
|                                           |
|assertion_COMPLETED='1'b;                  |
|                                           |
|END;                                       |
|                                           |
+-------------------------------------------+
```

where "function-name" is the name of the system-provided function and "function-name_COMPLETED" is the flag that is set by the conditional function upon its completion. Assertion_COMPLETED is set to '1' when the function completes its operation in order to trigger the execution of all nodes dependent on the completion of this node. To carry this out, conditional code before each dependent node is generated in the form:

```
+-------------------------------------------+
|                                           |
|IF assertion_COMPLETED THEN...             |
|                                           |
+-------------------------------------------+
```

This is generated by virtue of the previous creation of a flowchart entry for conditionals (refer back to Section 4.4.3.5 for generation of these flowchart table entries).

For example, if an assertion called "SUMX" were

```
+-------------------------------------------+
|                                           |
|Y=SUMMAT(X,...)                            |
|                                           |
+-------------------------------------------+
```

then the corresponding flowchart entry input to this routine would be

Node#	Node Type	Assn Name	CondFcn	Rplab	Text
n	ASSN	SUMX	SUMMAT	-	Y=SUMMAT (X...)

In such a case, this "Generate Procedure Code" routine would not only generate the PL/1 and its CALL statement, but also the control code to test for the function's completion and set the trigger variable to indicate the node's completion:

```
IF SUMMAT_COMPLETED THEN DO;

SUMMAT_COMPLETED='0'B;

SUMX_COMPLETED='1'; END;
```

Furthermore, the conditional flowchart table entry then causes generation of code to test the trigger variable before each dependent node (the determination of dependent nodes was explained in Section 4.4.3.5). In the above example, where the assertion name is SUMX, the following would be generated before dependent nodes:

```
IF SUMX_COMPLETED THEN ...
```

In summary, the assertions are implemented as CALLS to PL/1 procedures with conditional control code being generated for testing competion of any system-provided functions that are conditional, when appropriate.

4.5.1.6 Generating Iterative Code

Whenever the current flowchart entry indicates that an iteration begins at the current node, PL/1 code is generated by the GENDO procedure to loop through each of the elements of the repeatedly occurring item. The scope of the iteration has already been determined in the scope and iteration analysis section, and "DO" and "END" type flowchart entries have already been created during flowchart generation at the point of the beginning and end, respectively, of each loop. The iterative code can be implemented by the PL/1 DO iterative statement generated by this routine as follows:

```
+------------------------------------------+
|                                          |
|DO FOREACH_i = 1 to n;                    |
|                                          |
+------------------------------------------+
```

where i is the name of the repeating group of the iteration and n is the number of elements in the repeating group, which is either a constant for fixed-occurring items, or "EXIST.x" for variably-occurring items. The latter is the value telling the number of members actually existing in item x, provided in another assertion.

4.5.1.7 Generating Replacement-Stacking Code

It was mentioned earlier that MODEL allows a specification of assertions to be restated with a replacement of one of its components by one or more of a list of alternatives. For example, in the DEPSALE problem, if a certain item was out-of-stock, then there is an attempt to fill the order by the first available of a list of substitute items. For this purpose, a system-provided "Replacement" function was introduced which had the form

```
Y=REPLACE(X)
```

The procedure resulting from this statement was that the list in X, was to be stacked and delivered one at a time to Y, with all assertions dependent on Y to be repeated until another route was taken (such as the completion of the order in the DEPSALE problem) or until the list of substitutions was exhausted or emptied.

The implementation of such a feature can best be implemented as a system-provided run-time function called "REPLACE". When this function is invoked as Y=REPLACE(X), it would have the following steps:

(1) the first time that it is invoked (indicated by a flag called ALRREPL) , a stack is created with all the X's.

(2) The top element of X is delivered to Y.

(3) If the stack is empty, then the EMPTY choice is taken. If none is provided by the user, then the system defaults to printing a standard message and a branch is made to process the next record.

Code for the REPLACE function, as well as for other system-provided functions, can be found in Appendix B along with code for the system modules in alphabetical order.

When the replacement takes place by virtue of such a CALL, code needs to be generated to branch to the first dependent node on Y as determined during flowchart generation. This code is generated as part of the GPROCCD algorithm which has already been shown in Section 4.5.1.4. The following PL/1 code is generated immediately after the CALL to the replacement:

```
IF WASREPL THEN

IF ~ CHOICE.EMPTY THEN DO; WASREPL='0'B;

GO TO n;

END;
```

where n is the label corresponding to the first node dependent on Y. This code can be interpreted as: if a replacement took place (the WASREPL flag is set by REPLACE to indicate that a replacement was done), and if there was a replacement left (CHOICE is not empty), then reset flag and go to try another loop with the replacement. In the DEPSALE problem, for example, when a substitute item is used, a branch is needed back to the point where the item ordered is processed.

Furthermore, code needs to be generated to empty the stack of replacements whenever the next replacement results in a choice other than another replacement. This is accomplished by resetting the flags shown above as follows:

```
WASREPL=0
ALRREPL=0
#STACK=0
```

The #STACK variable is the level in the STACK used by the REPLACE function (the STACK is implemented as an array). Resetting the #STACK index effectively empties the stack by making it available for reuse (see Section 4.5.1.9 on resetting variables). In cases where the REPLACE feature is used in multiple assertions, different stacks need to be generated each with a unique name, although this was not demonstrated here.

4.5.1.8 Generating Code for Implicit Field Assignments

The "Generate Code for Implicit Field Assignments" (GIMFLD) routine is invoked by the Generate PL/1 Code control routine when reading a flowchart table entry for implicit field associations. Recall that such a node in the flowchart table was created for implicit correspondence of fields in the absence of explicit assertions, using such associations as same-named items, "OLD" and "NEW" keywords, etc. Algorithm GIMFLD simply generates the corresponding PL/1 code shown below, and therefore it is omitted.

```
+----------------------------------------+
|                                        |
|Target field = implicit source field;   |
|                                        |
+----------------------------------------+
```

with these two items designated already in the flowchart table entry.

4.5.1.9 Generating Code for Resetting Switches and Flags

Upon reaching a "reset" type flowchart entry (as described in Section 4.4.3.9), code is generated to reset the indicated variable to zero as follows:

```
+----------------------------------------+
|                                        |
|X=0                                     |
|                                        |
+----------------------------------------+
```

where X is the variable to be reset. Recall that all choices, conditions, and replacement flags are reset at the end of the outermost loop before going back to read the next record. Variables associated with specific functions (such as totals), however, are not reset here, but rather by the function itself upon completion.

4.5.1.10 Generating Code for Subsets

Code needs to be generated for every SUBSET type assertions so that only records meeting the subset criterion are considered. If a SUBSET is given for a source file, then code needs to be generated to branch to read the next record if the subset condition is not met. If a SUBSET is given for a target file, then code needs to be written to circumvent the corresponding "write" statements if the condition is not met. The code is generated for SUBSET type assertions by virtue of certain entries already present in the flowchart table.

Recall from Algorithm IDASSN, Step 7 (Section 4.4.3.5) that if a SUBSET assertion was given for a source file (i.e. the assertion had a target of the form "SUBSET. Filename", where "filename" is the name of a source file), then flowchart table entries of types COND and GOTO were written after the entry for the source record. The code that is generated for the condtional test is the following:

```
    IF ¬SUBSET. Filename THEN GO TO r;
```

where r is the label of the READ for the next record.

For SUBSET assertions for target files, Algorithm IDIOCD Step 10 (Section 4.4.3.3) generated a COND type flowchart table entry prior to the flowchart entry for the record in order to test for the subset criterion before writing the record. Whenever a SUBSET is described for a target file, the following code is always generated immediately before the WRITE:

```
    IF ¬SUBSET. Filename THEN
```

Since this is generated immediately before the corresponding WRITE, the record is thereby not written if the criterion is not met.

4.5.1.11 Generating PL/1 Declarations

The "Generate PL/1 Declarations" routine (GPL1DCL) has the task of accepting as input, the Declarations Table as presented in Section 4.4.3.11, and produces, as output, the corresponding PL/1 declarations which in turn cause the PL/1 compiler to allocate storage for the data to be used by the object program. Algorithm GPL1DCL generates the declaration code from the Declaration Table.

Recall from Section 4.4.3.11 that the Declarations Table entry already contains all the information necessary in order to generate the PL/1 declarations here, including the name being declared, the type of name, the "level" within the object hierarchic data structure, and any other attributes necessary for the declaration.

If an entry in the Declarations Table is for a FILE name, a PL/1 declaration is generated for the file, depending on the file attributes, according to the chart in Table 4.10. Notice that a unique file name is needed and generated by suffixing the letter 'S', 'T', OR 'U' to the user-provided file name, depending on the direction of the information flow (source, target, or both). The rationale for this suffix has been explained in Sections 4.4.3.3 and 4.4.3.11. The PL/1 compiler uses the file declarations for building internal file control tables and for communicating with the operating system.

Algorithm GPL1DCL: Generate PL/1 Declarations

(Input Declaration Table as described in Section 4.4.3.11)

```
Step 1. Read next declaration entry, at end go to Step 9.
Step 2. If entry type = 'FL' then generate file declaration according to Table 4.10.
Step 3. If entry type = 'RC' then generate "DCL recname CHAR(n) [VAR];"
Step 4. If entry type = 'FR' then generate file name as highest level name in PL/1 hierarchical structure (See Table 4.11); if file is both source and target, generate "OLD" and "NEW" declarations (as in Table 4.11).
Step 5. If entry type = 'RR' then generate record name as second highest entry in PL/1 hierarchical structure (See Table 4.11)
Step 6. If entry type = 'GP' then generate "n GRP (m),"
Step 7. If entry type = 'FD' then generate "n fieldname fieldtype (m) [var];"
Step 8. Go to Step 1.
Step 9. Return.
```

<table>
<tr><th colspan="4">If file is</th><th>Then The Declaration is</th></tr>
<tr><th>Input</th><th>Output</th><th>Sequential</th><th>Indexed</th><th></th></tr>
<tr><td>X</td><td></td><td>X</td><td></td><td>DCL X||'S'FILE INPUT RECORD SEQUENTIAL</td></tr>
<tr><td></td><td>X</td><td>X</td><td></td><td>DCL X||'T' FILE OUTPUT RECORD SEQL;</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td></td><td>DCL X||'S' FILE INPUT RECORD SEQL;
DCL X||'T' FILE OUTPUT RECORD SEQL;</td></tr>
<tr><td>X</td><td></td><td></td><td>X</td><td>DCL X||'S' FILE INPUT KEYED INDEXED DIRECT,</td></tr>
<tr><td></td><td>X</td><td></td><td>X</td><td>DCL X||'T' FILE OUTPUT KEYED INDEXED DIRECT;</td></tr>
<tr><td>X</td><td>X</td><td></td><td>X</td><td>DCL X||'U' FILE UPDATE KEYED INDEXED DIRECT .</td></tr>
</table>

Table 4.10

Generation of PL/1 File Declaration

If the Declarations Table entry is for a RECORD name, then a PL/1 string is declared which is used by the PL/1 program as a program area buffer for I/O operations. The PL/1 declarations generated here is

```
+-------------------------------------------+
|                                           |
|DCL recname CHAR(n) [VAR];                 |
|                                           |
+-------------------------------------------+
```

where the "recname" is the name of the record.

The file and record names in the Declarations Table are also used to generate the PL/1 declarations for the beginning of the hierarchic data structure of the record. The schema of the chart in Table 4.11 indicates how the beginning of such a PL/1 hierarchic structure is generated according to the file attributes. Notice that if a file is both a source and a target file, both an "OLD" and "NEW" area is declared and allocated memory.

The other declarations in the PL/1 data structure are generated by this routine upon processing the Declarations Table entries for groups and fields. If the entry is for a group, then a declaration for the group name is simply generated at the current level in the tree (with the subordinate fields to follow) with possible repetition factor following, as shown below:

```
+-------------------------------------------+
|                                           |
|n GRP (m),                                 |
|                                           |
+-------------------------------------------+
```

where n is the level in the tree, GRP is the name of the group, and m is the number of repetitions, if any.

If file I/O mode is	Then Beginning of PL/1 Hierarchic Structure is the following:
Source Only or Target Only	DCL 1 file name, 2 record name, 3 .... 3 ....
Both Source and Target	DCL 1 OLD, 2 file name, 3 record name, 4 ... 4 ... 1 NEW LIKE OLD,

Table 4.11

Generation of PL/1 Hierarchic Record Structure

If the Declarations Table entry is for a field, then a field declaration is generated in PL/1 at the current level, with all the attributes that have been filled into the declarations table entry. The generated PL/1 field declaration has the following form:

```
+-------------------------------------------+
|                                           |
|n fieldname fieldtype (m) [var];           |
|                                           |
+-------------------------------------------+
```

where n is the tree level; fieldname is the name of the field; the field type is character (CHAR), binary (BIN), numeric (NUMERIC), or fixed decimal (FIXED); m is the length of the field; and VAR is the optional PL/1 attribute to indicate a maximum length to be allocated for a string.

Generation of the above declarations all take place in this routine by reading the Declarations Table entries one at a time and generating the corresponding PL/1 declarations code, which turns out to be a fairly straight-forward transformation.

4.5.1.12 Other Code-Generation Supporting Routines

Certain routines have been found to be useful to all the code generation routines.

The "WRite PL/1" routine (WRPL1) is called by each of the code generating routines in order to write out the PL/1 code. Two parameters are passed to this routine: the string of PL/1 code to be written and the output file to which it should be written. WRPL1 takes the string containing one or more generated PL/1 statements and it outputs the PL/1 statement in the format and syntax that the PL/1 compiler expects. It ensures that the statement fits in columns 2 to 72 of each card necessary for the statement produced and generates sequence numbers in columns 73 to 80 of each card image.

The WRite DeCLarations routine (WRDCL) does the same for writing PL/1 declarations and indents the declarations according to the level numbers for readability. It is called by GPL1DCL in order to write out each declaration. It is passed two parameters: the string containing the declaration, and the level in the tree. The file to which the declarations are written is PL1DCL.

4.5.2 Code Generation Summary

The "Code Generation Summary" routine (CGSUM) has the task of wrapping up the code generation phase and writing a report to the user.

First, the different files with the generated PL/1 program (PL1DCL, PL1ON, PL1EX, PL1PROC) are merged (by MERGPL1) into one object PL/1 file (PL1OBJ) which can be subsequently compiled. Secondly, a Code Generation Summary Report is written which lists the generated PL/1 program to the user, and prints out the total number of lines generated. While the PL/1 listing would not be of much use to the average MODEL user, it would provide a deeper understanding for the more sophisticated user or system programmer for insight or debugging. This is analogous to the way that a PL/1 compiler can list a pseudo-assembly language listing for the object program that it generates, which can be of occasional use to certain users.

This routine also generates a few lines of statistics about the generated program that might be useful for the user, including the number of PL/1 statements generated and the amount of computer time used to generate the program.

The result of this entire code generation process is thus a complete PL/1 program ready to be compiled by the PL/1 compiler.

4.6 Compilation and Execution of the Generated Program

The PL/1 program produced by the MODEL Processor is submitted to the PL/1 Optimizing Compiler for translation into the host machine language. Since the MODEL Processor replaces the high-level language programmer (in PL/1), the PL/1 Optimizing compiler was the target machine kept in mind during generation of PL/1 code by the Processor. This enabled the Processor to leave some of the low-level program-generation and optimization tasks to the eventual recipient, the PL/1 compiler. For example, there was no need to generate OPEN or CLOSE statements in the MODEL Processor because the PL/1 compiler generates the OPEN before the first I/O operation, and a CLOSE at the end. There was no need for the Processor to concern itself with low-level optimization problems such as elimination of common arithmetic sub-expressions in arithmetic statements, or simplification of logical expressions, since such optimization is performed by the compiler. More details on the tasks, facilities, and execution logic of the PL/1 Optimizing Compiler can be found in appropriate manuals [PL175]. The program design, code-generation, and optimization performed by the MODEL Processor, together with the program and machine-code level optimization of the PL/1 Optimizing Compiler should yield an efficient and reliable object program ready to be executed.

The MODEL Processor and the PL/1 Compiler could be invoked as a unit by the MODEL user through a catalogued procedure. Thus, there would never be a need for a user to view this as a two-stage translation process. A specification would be submitted at one end, and a resulting program ready for execution would result at the other end. The resulting program module would be re-usable as long as there was no change to the specification.

CHAPTER 5

CONCLUSIONS

The preceding chapters have described a non-procedural language, MODEL, for describing modules of an information system and a processor for generating programs automatically from specifications expressed in MODEL. The research, system, and methodologies presented here have demonstrated the feasibility of such an approach. It is expected that such a language and system could be an important step toward the automation of the software development process if given strong industrial level support, reliability, and funding.

While some work, refinement, and extensions are needed on systems such as MODEL, several benefits and conclusions are already evident from this research. Such a system clearly reduces the amount of expertise and time needed to generate today's typical data processing programs. As outlined in Chapter 3, since the MODEL language is non-procedural, it enables its user to describe data and their interrelationships by providing a set of descriptive statements that can appear in arbitrary order. As described in Chapter 4, the MODEL Processor, unlike conventional compilers, is able to deduce the sequence of events, and is able to check much of the user's logic by analyzing a specification for its completeness and consistency and by providing effective feedback. In addition, it produces the desired program complete with sequence and control logic, declarations, input/output commands, etc., relieving the user entirely of such procedural thinking. The amount of writing required of the user is certainly reduced as the ratio of number of statements in a typical MODEL specification to the number of generated program statements is approximately 1:4. But it is not so much the number of statements that is important as is the fact that the non-procedural and very high level nature of MODEL requires less expertise than would programming.

Comparing the program generated by the Processor with a manually written program, the execution time efficiency is good. Unlike "generalized packages," the Processor generates an ad hoc program for a particular problem. Therefore, the PL/1 code generated is peculiar to a given use and contains no unnecessary instructions. The programs generated by the Processor appear to be as concise, structured and efficient as good manually-written ones, though their structure does not necessarily parallel a human-written program, as explained in Chapter 4. In short, the MODEL language and system promise to produce fruitful and positive results if research on such a system is maintained and expanded.

There are several directions that further research may take in the future. On the pragmatic level, there are a number of refinements that could be made to make the current version of MODEL more attractive. The current syntax of the language is somewhat restrictive partly because it is a formal language, and partly because of various restrictions that were made, such as the decision to make arithmetic and logical expressions compatible with PL/1. A worthwhile endeavor would then be to make a more English-like and user-oriented syntax. In fact, a still more ambitious expansion would be to convert the

Processor to one with an on-line real-time capability. An interactive session with a user at a terminal through a natural language (using current state-of-the-art natural language techniques) would make the interaction between the user and the system much more effective. Since a system definer using the version of MODEL described here would typically go through several interactions until his problem is solved, an interactive terminal approach would certainly enhance this process, and such a possibility is surely one to be investigated.

There are various other extensions that could be made to MODEL, some minor and some major in scope. The current version supports the sequential and indexed sequential access methods, so it would be useful to extend the support to other and more complex data organizations and access systems. The library of functions could be enlarged to encompass more functions that would be useful in a typical data processing environment. The Processor could be extended to incorporate more elaborate report generation facilities as found in commercial report generators. The target language could be supplemented with a choice to generate COBOL programs rather than PL/1. Finally, the efficiency of some of the algorithms used in the Processor could be improved. For example, the matrix manipulation procedures should use sparse matrix techniques since many of the entries are zero.

On a more research-oriented and significant plane, MODEL itself could be extended upwards in the ladder of phases of the software development life cycle. In other words, MODEL could be extended so that the user would not have to separate the data set into files and not have to describe physical media. The Processor would then have to be extended to perform these decisions automatically, by incorporating some of the automatic physical design techniques reviewed in Chapter 2. Of course, the most challenging work still remains in attempting to automate the production of functional specifications themselves by using problem solving models and interaction in natural language, but such progress will have to wait for future years.

In summary, the MODEL Language and Processor have made one step in the road to total automation of the software development process. While much research in this field lies ahead, it is hoped that this research has made a contribution towards that end and has aided the effective utilization of manpower and machine.

APPENDIX A

AN EXAMPLE OF THE USE OF MODEL

DESCRIPTION OF A SALE FUNCTION IN A DEPARTMENT STORE

The purpose of this appendix is to provide a complete example of the use of the MODEL language and Processor. The example presented here is the Department Store Sale problem (DEPSALE) which has been referenced throughout the dissertation, and segments of which have been used as examples throughout Chapters 3 and 4. The environment of this example is a department store with many departments, a large number of charge account customers, and a diverse stock inventory. Point-of-sale terminals, connected to a network of computers, are distributed throughout the several locations of the department store.

The function that the analyst wishes to describe here involves purchases made by charge account customers. It is desired to have a computer program which will perform the charge accounting and sale functions. The objective of the following example is to show how this Department Store Sale problem (DEPSALE) is described in MODEL and to exemplify how the MODEL Processor would subsequently process this specification and automatically generate a program to perform the function.

Figure A1 is an illustration of the DEPSALE function, as presented in Chapter 3. It shows the function at the center of the figure, with all the interacting data. The source data for DEPSALE are three files: the sales transactions which come from a terminal (TRANS), sequentially, one at a time, and which contain the information provided by the purchaser. There is a customer master file (CUSTMAST) which uses a disk as a storage medium and where records of customers could be referenced by providing customer numbers. Finally, there is an inventory file (INVEN), which also uses disk storage medium and where information on stock items could be referenced by providing a stock number.

The target data are the updated records in CUSTMAST and INVEN affected by the sale transaction. An entry is also made in a sales journal (JOURN), a sequential output file. Finally, a sales slip (SALESLIP) is produced on the terminal, if the sales transaction has been consumated, or alternately an exception notice (EXCEPT), if for some reason the transaction cannot take place.

Chapter 3 has already presented the sections and components of the MODEL language in detail. To review, a MODEL specification consists of three major parts: the header, the data descriptions, and the statements specific to the module. The header contains information identifying the module name, the source files, and the target files.

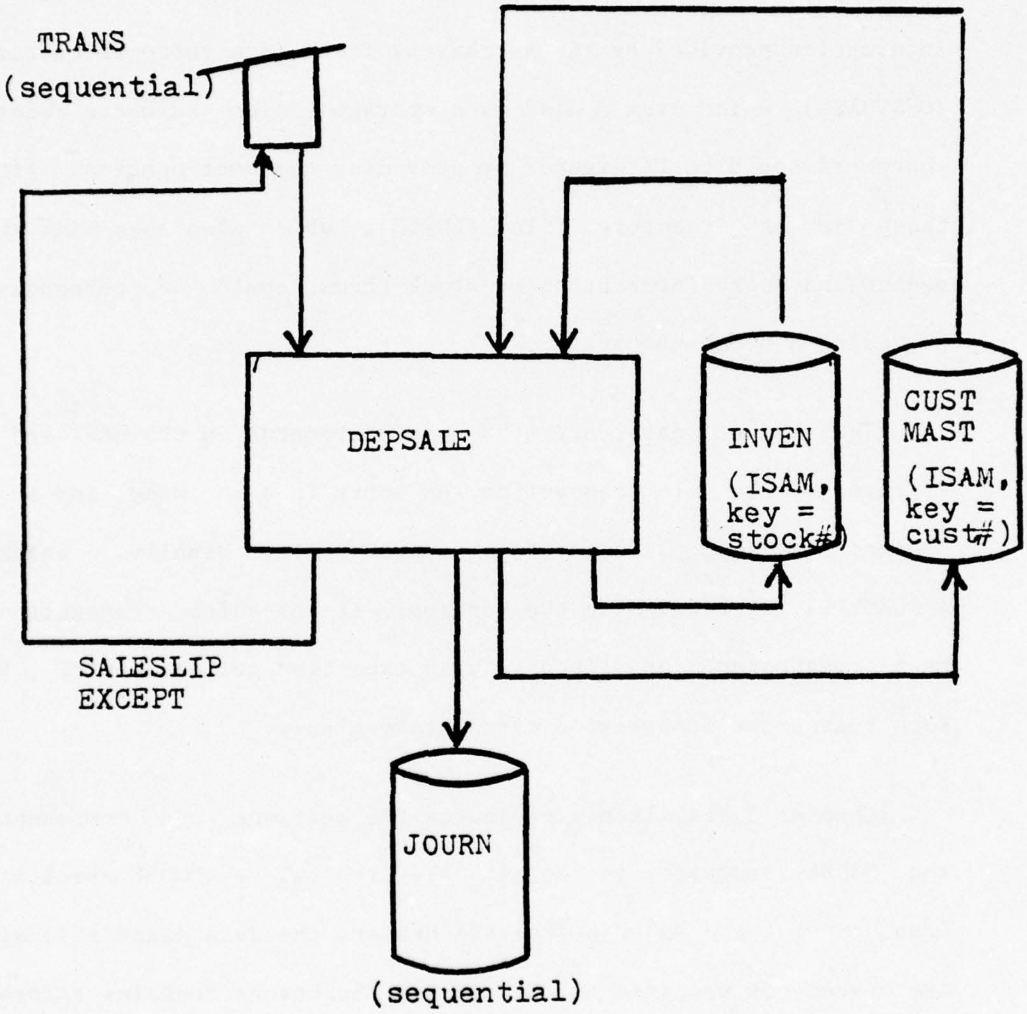


Figure A1: Illustration of Department Store Sale

The data descriptions describe the structure, format, and attributes of the files and their component records and data elements. This section is independent of the module description since data can be used by several modules. In addition to description of the data, this section may also be used to provide media descriptions and a set of assertions that may be used dynamically to evaluate data dependent structures such as variable length and repeating data. Figure A2 shows a data network for the DEPSALE function. for instance, at the top left is a tree structure showing the component fields of a tree structure. The arrows in the center of the diagram indicate the inter-file relationships between the TRANS, INVEN, and CUSTMAST files. The customer number in the sales transaction can be used to access the corresponding customer record, while the stock number can be used to access the corresponding inventory record. Furthermore, the SUBST field contains a list of stock numbers which could be used as substitutes for the main stock number. The direction of the pointer is such that for any of the source records one could find the corresponding target records. In a similar manner the other files, their components, and their interrelationships are entered in the network.

The assertion section of MODEL allows assertions of various types to be made. The source set and target set assertions are used to specify subsets of files to be processed or produced. Other types of assertions are used to express decision rules, formulae, and other relationships. The "source" and "target" headings of each assertion

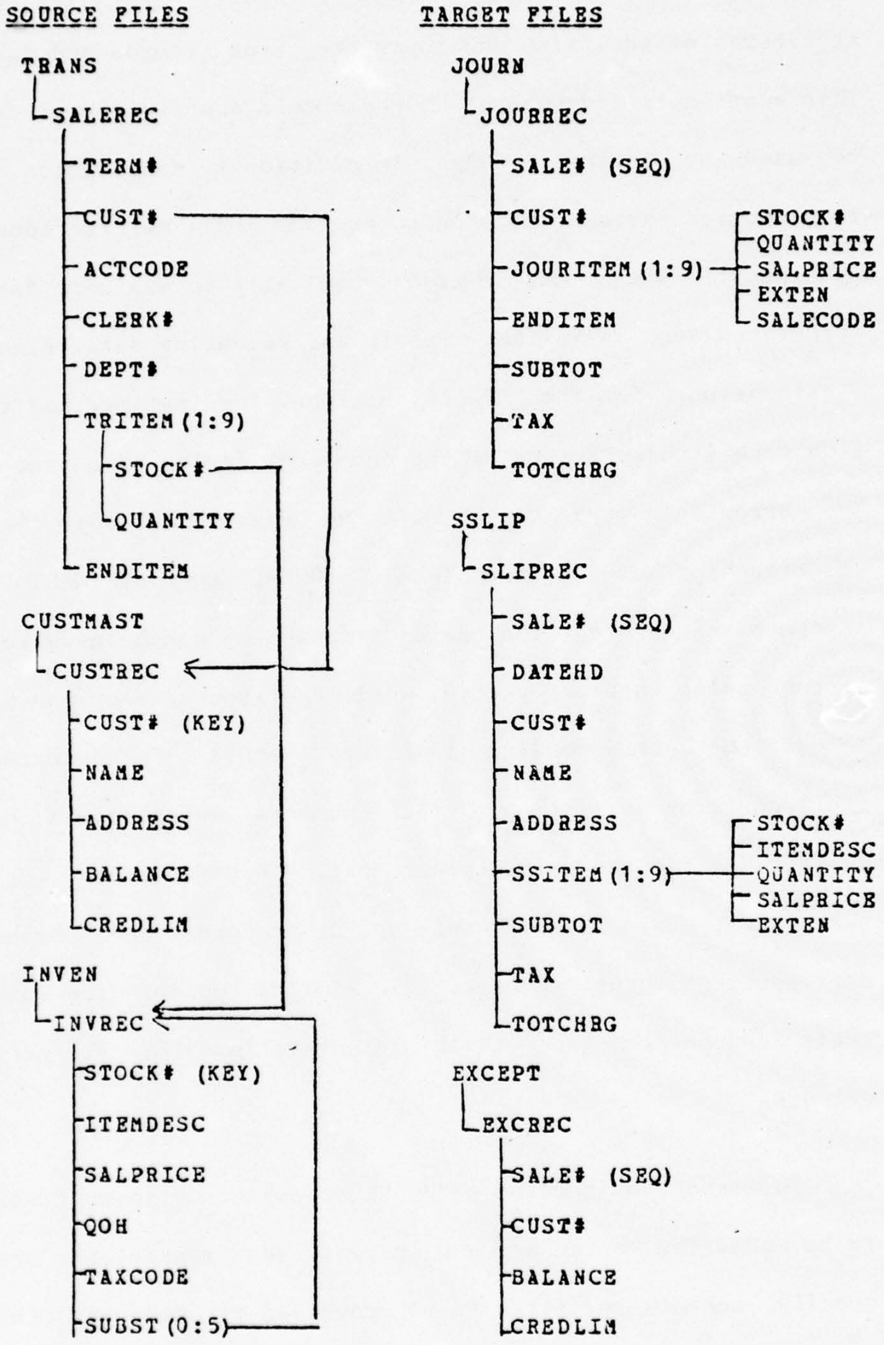


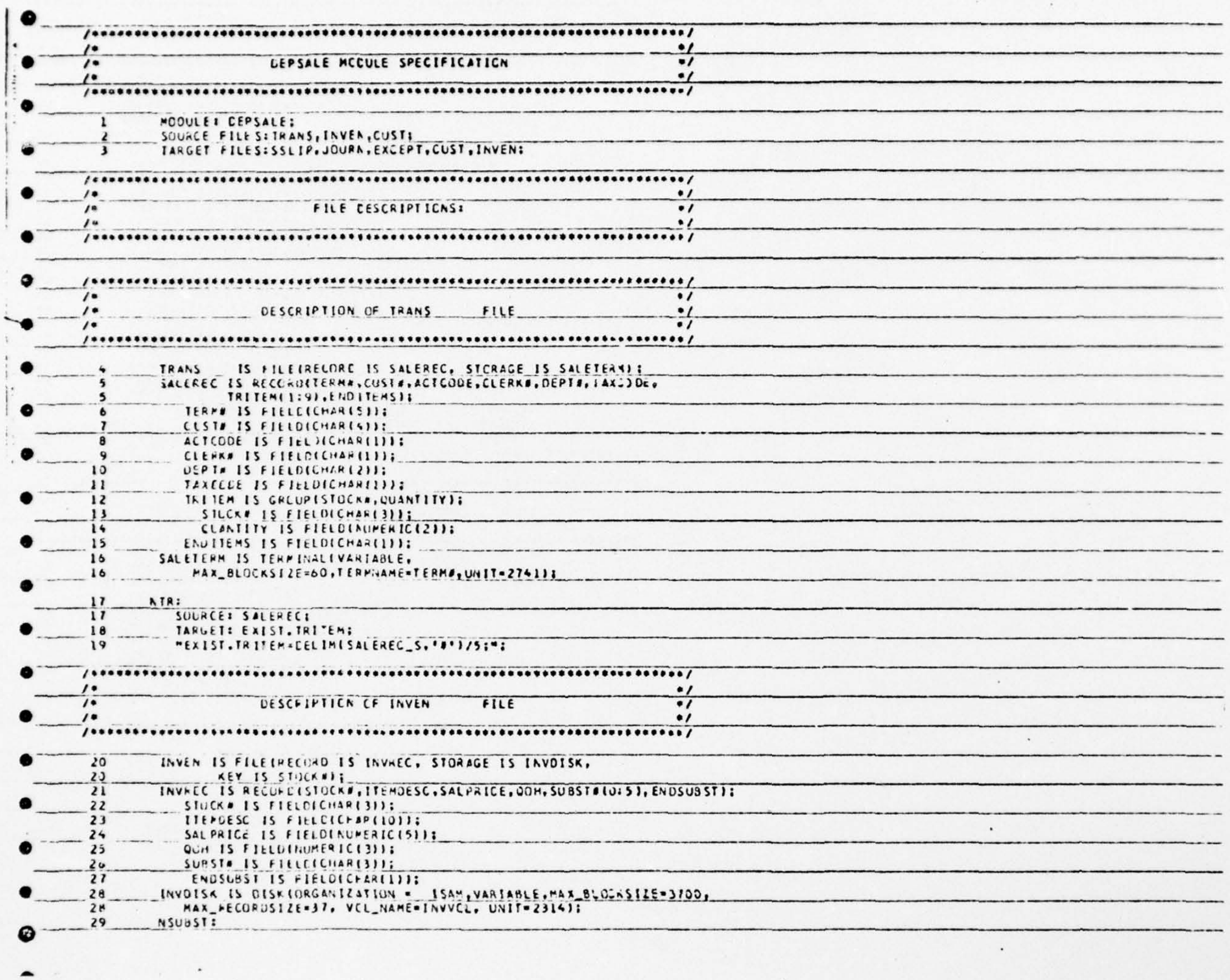
Figure A2: Data Network for DEPSALE Problem

can be viewed as a temporary measure to facilitate implementation.

Figure A3 is the complete example of the department store sale problem in MODEL. At the very top the header information is provided: the module name (DEPSALE) and names of the source and target files. Then each one of the files is described separately in greater detail. The files are progressively subdivided into records, groups, and fields using the network diagram described above. Where data is fixed length, the number of characters or digits is provided. Where data is variable length, or where the number of repetitions varies, or is optional, assertions are provided at the end of the file description. For instance, there may be a variable number (1 to 9) of items in a sales transaction, and the number is determined by the delimiting character as indicated by the DELIM function described in the assertion named NTR. In the Interfile Relationships section, the two pointers from the sales transaction file to the INVEN and CUSTMAST files as shown in the network figure are described in respective assertions.

Finally, in the assertions section, several decision and accounting rules are given. The first four show how to compute an extension, how to compute the sub-total and tax, and how to compute the total charge to the customer. The next two rules (TRSALE and TRSUB) determine whether there is sufficient quantity on hand of the item ordered or whether a substitution needs to be tried. The next rule (TRYREPL) indicates that if the desired item is out of stock, an attempt should be made to complete the transaction by providing a substitute (SUBST) for the desired item. The next five rules define the various values of the sale slip and journal. Another rule (EXLIM) determines whether the customer exceeded his credit limit, and if so, sends a message (ERROR1). The next two rules (UPDQUANT and ADJ-BALC) show how the quantity on hand and balance are to be updated. The next two rules (CALCSAL# and SLIPDATE) utilize built-in functions to generate serial numbers and the date respectively. Finally, the last two assertions provide target set criteria for the exception report and saleslip.

Figure A3 shows the complete listing of the DEPSALE problem in the MODEL language. Chapter 4 has explained in great detail how the MODEL Processor analyzes such a specification syntactically and semantically, storing the statements in an associative memory, how it represents and analyzes relationships in the MODEL specification in a matrix representing a directed graph, how it produces various reports, and how it generates the desired program. In addition to the MODEL specification, Figure A3 presents the cross-reference report and the matrix report that would be presented by the MODEL Processor for the DEPSALE problem. Furthermore, Figure A4 shows the network of relationships for this problem in pictorial form, for which the Processor uses matrix representation.



```
        /**************************************************************/
        /*                                                            */
        /*               DEPSALE MODULE SPECIFICATION                 */
        /*                                                            */
        /**************************************************************/

    1        MODULE: DEPSALE;
    2        SOURCE FILES:TRANS,INVEN,CUST;
    3        TARGET FILES:SSLIP,JOURN,EXCEPT,CUST,INVEN;

        /**************************************************************/
        /*                                                            */
        /*                    FILE DESCRIPTIONS:                      */
        /*                                                            */
        /**************************************************************/

        /**************************************************************/
        /*                                                            */
        /*              DESCRIPTION OF TRANS      FILE                */
        /*                                                            */
        /**************************************************************/

    4        TRANS    IS FILE(RECORD IS SALEREC, STORAGE IS SALETERM);
    5        SALEREC IS RECORD(TERM#,CUST#,ACTCODE,CLERK#,DEPT#,TAXCODE,
    5                 TRITEM(1:9),ENDITEMS);
    6          TERM# IS FIELD(CHAR(5));
    7          CUST# IS FIELD(CHAR(4));
    8          ACTCODE IS FIELD(CHAR(1));
    9          CLERK# IS FIELD(CHAR(1));
   10          DEPT# IS FIELD(CHAR(2));
   11          TAXCODE IS FIELD(CHAR(1));
   12          TRITEM IS GROUP(STOCK#,QUANTITY);
   13            STOCK# IS FIELD(CHAR(3));
   14            QUANTITY IS FIELD(NUMERIC(2));
   15          ENDITEMS IS FIELD(CHAR(1));
   16       SALETERM IS TERMINAL(VARIABLE,
   16           MAX_BLOCKSIZE=60,TERMNAME=TERM#,UNIT=2741);

   17      NTR:
   17         SOURCE: SALEREC;
   18         TARGET: EXIST.TRITEM;
   19         "EXIST.TRITEM=DELIM(SALEREC_S,'#')/5;";

        /**************************************************************/
        /*                                                            */
        /*              DESCRIPTION OF INVEN      FILE                */
        /*                                                            */
        /**************************************************************/

   20        INVEN IS FILE(RECORD IS INVREC, STORAGE IS INVDISK,
   20               KEY IS STOCK#);
   21        INVREC IS RECORD(STOCK#,ITEMDESC,SALPRICE,QOH,SUBST#(0:5),ENDSUBST);
   22           STOCK# IS FIELD(CHAR(3));
   23           ITEMDESC IS FIELD(CHAR(10));
   24           SALPRICE IS FIELD(NUMERIC(5));
   25           QOH IS FIELD(NUMERIC(3));
   26           SUBST# IS FIELD(CHAR(3));
   27            ENDSUBST IS FIELD(CHAR(1));
   28        INVDISK IS DISK(ORGANIZATION =  ISAM,VARIABLE,MAX_BLOCKSIZE=3700,
   28           MAX_RECORDSIZE=37, VOL_NAME=INVVOL, UNIT=2314);
   29       NSUBST:
```

Figure A3 Listing of DEPSALE Problem in MODEL

```
29        SOURCE: OLD.INVREC;
30        TARGET: EXIST.SUBST#;
31      "EXIST.SUBST#=DELIM(OLD_INVREC_S,'#')1/31";

     /*************************************************************/
     /*                                                           */
     /*               DESCRIPTION OF CUST      FILE               */
     /*                                                           */
     /*************************************************************/

32      CUST      IS FILE(RECORD IS CUSTREC, STORAGE IS CUSTDISK,
32              KEY IS CUST#);
33      CUSTREC IS RECORD(CUST#, NAME, ADDRESS, BALANCE, CREDLIM);
34          CUST# IS FIELD(CHAR(4));
35          NAME IS FIELD(CHAR(20));
36          ADDRESS IS FIELD(CHAR(40));
37          BALANCE IS FIELD(NUMERIC(5));
38          CREDLIM IS FIELD(NUMERIC(5));
39      CUSTDISK IS DISK(ORGANIZATION = ISAM, FIXED, BLOCKSIZE=740,
39          RECORDSIZE=74, VOL_NAME=CUSTVOL, UNIT=2314);

     /*************************************************************/
     /*                                                           */
     /*               DESCRIPTION OF SSLIP      REPORT            */
     /*                                                           */
     /*************************************************************/

40      SSLIP     IS REPORT(REPORT_ENTRY IS SLIPREC, STORAGE IS SLIPTERM,
40          SEQUENCE IS SALES#);
41      SLIPREC IS REPORT_ENTRY(SALES#,DATEHD,CUST#,NAME,ADDRESS,
41                  SSITEM(1:9),ENDITEMS,SUBTOT,TAX,TOTCHRG);
42          SALES# IS FIELD(NUMERIC(4));
43           DATEHD IS FIELD(CHAR(13));
44          CUST# IS FIELD(CHAR(4));
45          NAME IS FIELD(CHAR(20));
46          ADDRESS IS FIELD(CHAR(40));
47          SSITEM IS GROUP(STOCK#,ITEMDESC,QUANTITY,SALPRICE,EXTEN);
48            STOCK# IS FIELD(CHAR(3));
49            ITEMDESC IS FIELD(CHAR(10));
50            SALPRICE IS FIELD(NUMERIC(5));
51            EXTEN IS FIELD      (NUMERIC(8));
52            QUANTITY IS FIELD(NUMERIC(2));
53          ENDITEMS IS FIELD(CHAR(1));
54           SUBTOT IS FIELD(NUMERIC(8));
55          TAX IS FIELD(NUMERIC(5));
56          TOTCHRG IS FIELD(NUMERIC(10));
57      SLIPTERM IS TERM(NAL(VARIABLE,
57           MAX_BLOCKSIZE=357,TERMNAME=TERM#,UNIT=2741);

58     ASS:
58        SOURCE: EXIST.TRITEM;
59        TARGET: EXIST.SSITEM;
60       "EXIST,SSITEM=EXIST.TRITEM;";

     /*************************************************************/
     /*                                                           */
     /*               DESCRIPTION OF JOURN      FILE              */
     /*                                                           */
     /*************************************************************/

61      JOURN     IS FILE(RECORD IS JOURREC, STORAGE IS JOURDISK,
61          SEQUENCE IS SALES#);
62      JOURREC IS RECORD(SALES#,CUST#,JOURITEM(1:9),ENDITEMS,SUBTOT,
62              TAX,TOTCHRG,CREDCODE);
63          SALES# IS FIELD(NUMERIC(4));
64          CUST# IS FIELD(CHAR(4));
65          JOURITEM IS GROUP(STOCK#,QUANTITY,SALPRICE,EXTEN,SALECODE);
66            STOCK# IS FIELD(CHAR(3));
67            QUANTITY IS FIELD(NUMERIC(2));
68            SALPRICE IS FIELD(NUMERIC(5));
69            EXTEN IS FIELD(NUMERIC(8));
70            SALECODE IS FIELD(CHAR(2));
71          ENDITEMS IS FIELD(CHAR(1));
72           SUBTOT IS FIELD(NUMERIC(8));
73          TAX IS FIELD(NUMERIC(5));
74          TOTCHRG IS FIELD(NUMERIC(10));
75          CREDCODE IS FIELD(CHAR(2));

76      JOURDISK IS DISK(VARIABLE, MAX_BLOCKSIZE=2140,
75           MAX_RECORDSIZE=214,VOL_NAME=JOURVOL,UNIT=2314);

77      AJR:
77         SOURCE: EXIST.TRITEM;
78         TARGET: EXIST.JOURITEM;
79         "EXIST.JOURITEM=EXIST.TRITEM;";

/*************************************************************/
/*                                                           */
/*              DESCRIPTION OF EXCEPT     REPORT             */
/*                                                           */
/*************************************************************/

80      EXCEPT IS REPORT(REPORT_ENTRY IS EXCREC, STORAGE IS EXCTERM,
80         SEQUENCE IS SALES#);
81      EXCREC IS REPORT_ENTRY(SALES#,CUST#,BALANCE,CREDLIM,CREDCODE);
82         SALES# IS FIELD(NUMERIC(4));
83         CUST# IS FIELD(CHAR(4));
84         BALANCE IS FIELD(NUMERIC(5));
85         CREDLIM IS FIELD(NUMERIC(5));
86         CREDCODE IS FIELD(CHAR(2));

87      EXCTERM IS TERMINAL(FIXED,
87          BLOCKSIZE=20,TERMNAME=TERM#,UNIT=2741);

/*************************************************************/
/*                                                           */
/*                 INTERFILE RELATIONSHIPS                   */
/*                                                           */
/*************************************************************/

88      INTERFILE RELATIONSHIPS:
88       TR_CUST:
88         SOURCE: TRANS.CUST#;
89         TARGET: POINTER.OLD.CUSTREC;
90         "POINTER.OLD.CUSTREC=TRANS.CUST#;" ;

91       TR_INV:
91         SOURCE:    TRANS.STOCK#(FOREACH_TRITEM);
92         TARGET: POINTER.OLD.INVREC;
93         "POINTER.OLD.INVREC=   TRANS.STOCK#(FOREACH_TRITEM);";


/*************************************************************/
/*                                                           */
/*                  MODULE DESCRIPTION:                      */
```

```
      /*                                                              */
      /****************************************************************/

      /****************************************************************/
      /*                                                              */
      /*                    INTERIM DESCRIPTIONS                      */
      /*                                                              */
      /****************************************************************/

 94   INTERIM DESCRIPTIONS;
 94     SALES# IS INTERIM(NUMERIC(4));

      /****************************************************************/
      /*                                                              */
      /*                    ASSERTIONS SECTION                        */
      /*                                                              */
      /****************************************************************/

 95   ASSERTIONS SECTION;
 95     CALC_EXT:
 95       SOURCE: OLD.INVEN.SALPRICE,     TRANS.QUANTITY(FOREACH_TRITEM);
 96       TARGET:     SSLIP.EXTEN(FOREACH_TRITEM);
 97         "SSLIP.EXTEN(FOREACH_TRITEM)=OLD.INVEN.SALPRICE *
 97              TRANS.QUANTITY(FOREACH_TRITEM);";

 98     CALCSTCT:
 98         SOURCE: SSLIP.EXTEN,CHOICE.SALE;
 99         TARGET: SSLIP.SUBTCT;
100         "IF CHOICE.SALE=SELECTED THEN  SSLIP.SUBTOT=SUM(SSLIP.EXTEN,
100         FOREACH_TRITEM,1,EXIST.TRITEM);";

101     CALC_TAX:
101       SOURCE: TAXCODE,SSLIP.SUBTOT,CHOICE.SALE;
102       TARGET: SSLIP.TAX;
103       "IF CHOICE.SALE=SELECTED THEN
103         IF TAXCODE='N' THEN SSLIP.TAX=0;
103         ELSE SSLIP.TAX=.05*    SSLIP.SUBTOT ;";

104     CALCCHRG:
104       SOURCE: SSLIP.SUBTOT,SSLIP.TAX;
105       TARGET:SSLIP.TOTCHRG;
106         "SSLIP.TOTCHRG=SSLIP.SUBTOT+SSLIP.TAX;";

107     TRSALE:
107       SOURCE:     TRANS.QUANTITY(FOREACH_TRITEM),OLD.INVEN.QOH;
108       TARGET: CHOICE.SALE;
109         " IF     TRANS.QUANTITY(FOREACH_TRITEM)<= OLD.INVEN.QOH THEN
109                CHOICE.SALE=SELECTED;     ELSE CHOICE.SALE=NOT_SELECTED;";

110     TRSUB:
110       SOURCE:     TRANS.QUANTITY(FOREACH_TRITEM),OLD.INVEN.QOH;
111       TARGET: CHOICE.SUBSTIT;
112         " IF     TRANS.QUANTITY(FOREACH_TRITEM) > OLD.INVEN.QOH THEN
112         CHOICE.SUBSTIT=SELECTED; ELSE CHOICE.SUBSTIT=NOT_SELECTED;";

113     TRYREPL:
113       SOURCE: OLD.INVEN.SUBST#,CHOICE.SUBSTIT,EXIST.SUBST#;
114       TARGET: POINTER,OLD.INVREC;
115       FUNCTION: REPLACE;
116         "IF CHOICE.SUBSTIT=SELECTED THEN POINTER.OLD.INVREC =
116            REPLACE(OLD.INVEN.SUBST#,EXIST.SUBST#);";
```

```
117     RECHOLR:
117       SOURCE:CHOICE.EMPTY,CHOICE.SUBSTIT;
118       TARGET:     JOURN.SALECODE(FOREACH_TRITEM);
119        "IF CHOICE.SUBSTIT=SELECTED & CHOICE.EMPTY=SELECTED
119            THEN JOURN.SALECODE(FOREACH_TRITEM)='RO';
119           ELSE    JOURN.SALECODE(FOREACH_TRITEM)='OK';";

120     SLIPRICE:
120       SOURCE: OLD.INVEN.SALPRICE;
121       TARGET: SSLIP.SALPRICE(FOREACH_TRITEM);
122       "SSLIF.SALPRICE(FOREACH_TRITEM)=OLD.INVEN.SALPRICE;";

123     SLIPDESC:
123       SOURCE: OLD.INVEN.ITEMDESC;
124       TARGET: SSLIP.ITEMDESC(FOREACH_TRITEM);
125       "SSLIP.ITEMDESC(FOREACH_TRITEM)=OLD.INVEN.ITEMDESC;";

126     JRPRICE:
126       SOURCE: OLD.INVEN.SALPRICE;
127       TARGET: JOURN.SALPRICE(FOREACH_TRITEM);
128       "JOURN.SALPRICE(FOREACH_TRITEM)=OLD.INVEN.SALPRICE;";

129     NCREPL:
129       SOURCE:CHOICE.EMPTY,CHOICE.SUBSTIT;
130       TARGET:CHOICE.SALE;
131         "IF CHOICE.EMPTY = SELECTED & CHOICE.SUBSTIT=SELECTED THEN
131         CHOICE.SALE=SELECTED;" ;

132     EXLIM:
132       SOURCE:OLD.CUST.BALANCE,SSLIP.TOTCHRG,OLD.CUST.CREDLIM)
133      TARGET: CHOICE.EXCRLIM;
134         "IF OLD.BALANCE +SSLIP.TOTCHRG > OLD.CUST.CREDLIM THEN
134         CHOICE.EXCRLIM=SELECTED;";
135     ERRORL:

135       SOURCE: CHOICE.EXCRLIM;
136       TARGET: JOURN.CREDCODE;
137       "IF CHOICE.EXCRLIM = SELECTED  THEN    JOURN.CREDCODE='CR';
137           ELSE    JOURN.CREDCODE='OK';";

138     UPDQUANT:
138       SOURCE: CHOICE.SALE, OLD.INVEN.QOH,    TRANS.QUANTITY(FOREACH_TRITEM);
139       TARGET: NEW.QOH;
140       "IF CHOICE.SALE = SELECTED  THEN NEW.QOH =
140       OLD.QOH-TRANS.QUANTITY(FOREACH_TRITEM);";

141     ADJ_BALC:
141       SOURCE: CHOICE.SALE, OLD.CUST.BALANCE,SSLIP.TOTCHRG,CHOICE.SUBSTIT;
142       TARGET: NEW.CUST.BALANCE;
143       "IF CHOICE.SALE = SELECTED  THEN NEW.CUST.BALANCE=
143       OLD.CUST.BALANCE+SSLIP.TOTCHRG;";

144     CALLSAL#:
144       SOURCE: SALEREC;
145       TARGET:INTERIM.SALES#;
146       FUNCTION: SERIAL#;
147       "INTERIM.SALES#=SERIAL#(1,1);" ;

148     SLIPDATE:
148         TARGET:SSLIP.DATEHC;
149       FUNCTION: TODAY;
```

```
150             "SSLIP.CATEHO=TCCAY;";

151       TARSET1:
151          SOURCE: CHOICE.SALE;
152          TARGET:SUBSET.SSLIP;
153          "IF CHOICE.SALE=SELECTED THEN
153          SUBSET.SSLIP=SELECTED;
153          ELSE SUBSET.SSLIP=   NOT_SELECTED;";

154       TARSET2:
154          SOURCE: CHOICE.EXCRLIM;
155          TARGET: SUBSET.EXCEPT;
156          "IF CHOICE.EXCRLIM=SELECTED THEN SUBSET.EXCEPT=SELECTED;
155          ELSE SUBSET.EXCEPT=NOT_SELECTED;";
157          END;
```

```
THE FOLLOWING SYNTACTIC ERRORS AND WARNINGS DETECTED:

(NONE)
```

```
                                         CROSS REFERENCE AND ATTRIBUTES REPORT

      NAME           STATEMENT           ATTRIBUTES                     REFERENCES
                     DESCRIPTION

    ACTCODE               8              FIELD, CHARACTER IN FILE          5
                                         TRANS
    ADDRESS              36              FIELD, CHARACTER IN FILE         33
                                         CUST
    ADDRESS              46              FIELD, CHARACTER IN FILE         41
                                         SSLIP
    ADJ_BALC            143              ASSERTION
    BALANCE              37              FIELD, NUMERIC IN FILE           33,132,141,143
                                         CUST
    BALANCE              84              FIELD, NUMERIC IN FILE           81
                                         EXCEPT
    CALC_EXT             97              ASSERTION
    CALC_TAX            103              ASSERTION
    CALCCHRG            106              ASSERTION
    CALCSAL#            147              ASSERTION
    CALCSTOT            100              ASSERTION
    CHOICE                               RESERVED WORD                    98,101,109,112,113,117,129,131,134,135,138,141,
                                                                          154
    CLERK#                9              FIELD, CHARACTER IN FILE          5
                                         TRANS
    CREDCODE             75              FIELD, CHARACTER IN FILE         62,137
                                         JOURN
    CREDCODE             86              FIELD, CHARACTER IN FILE         81
                                         EXCEPT
    CREDLIM              38              FIELD, NUMERIC IN FILE           33,132
                                         CUST
    CREDLIM              85              FIELD, NUMERIC IN FILE           81
                                         EXCEPT
    CUST                 32              FILE,SOURCE,TARGET,               2,  3,132,141,143
                                         SORTED/KEYED
    CUST#                 7              FIELD, CHARACTER IN FILE          5, 88
                                         TRANS
    CUST#                34              FIELD, CHARACTER IN FILE         32, 33
                                         CUST
    CUST#                44              FIELD, CHARACTER IN FILE         41
                                         SSLIP
    CUST#                64              FIELD, CHARACTER IN FILE         62
                                         JOURN
    CUST#                83              FIELD, CHARACTER IN FILE         81
                                         EXCEPT
    CUSTDISK             39              DISK NAME                        32
    CUSTREC              33              RECORD,( 5 SUB-MEMBERS),         32, 90
                                         IN FILE CUST
    DATEPD               43              FIELD, CHARACTER IN FILE         41,150
                                         SSLIP
    DEPSALE               1              MODULE NAME
    DEPT#                10              FIELD, CHARACTER IN FILE          5
                                         TRANS
    EMPTY                                RESERVED WORD                   117,129
    ENDITEMS             15              FIELD, CHARACTER IN FILE          5
                                         TRANS
    ENDITEMS             53              FIELD, CHARACTER IN FILE         41
                                         SSLIP
    ENDITEMS             71              FIELD, CHARACTER IN FILE         62
                                         JOURN
```

```
ENDSUBST     27      FIELD, CHARACTER IN FILE     21
                     INVEN
ERROR1      137      ASSERTION
EXCEPT       80      REPORT,TARGET,                3,156
                     SORTED/KEYED
EXCREC       81      REPORT_ENTRY,( 5             80
                     SUB-MEMBERS),IN FILE
                     EXCEPT
EXCRLIM      81      CHOICE NAME                  134,135,154
EXCTERM      87      TERMINAL NAME                80
EXIST                RESERVED WORD                19, 31, 58, 60, 77, 79,113
EXLIM       134      ASSERTION
EXTEN        51      FIELD, NUMERIC IN FILE       47, 97, 98
                     SSLIP
EXTEN        69      FIELD, NUMERIC IN FILE       65
                     JOURN
INTERIM              RESERVED WORD                147
INVDISK      28      DISK NAME                    20
INVEN        20      FILE,SOURCE,TARGET,           2,  3, 95,107,110,113,120,123,126,138
                     SORTED/KEYED
INVREC       21      RECORD,( 6 SUB-MEMBERS),     20, 29, 93,116
                     IN FILE INVEN
ITEMDESC     23      FIELD, CHARACTER IN FILE     21,123
                     INVEN
ITEMDESC     49      FIELD, CHARACTER IN FILE     47,125
                     SSLIP
JOURDISK     76      DISK NAME                    61
JOURITEM     65      GROUP,( 5 SUB-MEMBERS),      62, 79
                     IN FILE JOURN
JOURN        61      FILE,TARGET,SORTED/KEYED      3,119,128,137
JOURREC      62      RECORD,( 8 SUB-MEMBERS),     61
                     IN FILE JOURN
JRPRICE     128      ASSERTION
NAME         35      FIELD, CHARACTER IN FILE     33
                     CUST
NAME         45      FIELD, CHARACTER IN FILE     41
                     SSLIP
NEW                  RESERVED WORD                140,143
NJR          79      ASSERTION
NOREPL      131      ASSERTION
NSS          60      ASSERTION
NSUBST       31      ASSERTION
NTR          19      ASSERTION
OLD                  RESERVED WORD                29, 90, 93, 95,107,110,113,116,120,123,126,132,
                                                  141
POINTER              RESERVED WORD                90, 93,116
QOH          25      FIELD, NUMERIC IN FILE       21,107,110,138,140
                     INVEN
QUANTITY     14      FIELD, NUMERIC IN FILE       12, 95,107,110,138
                     TRANS
QUANTITY     52      FIELD, NUMERIC IN FILE       47
                     SSLIP
QUANTITY     67      FIELD, NUMERIC IN FILE       65
                     JOURN
REORDER     119      ASSERTION
SALE        119      CHOICE NAME                  98,101,109,131,138,141,151
SALECODE     70      FIELD, CHARACTER IN FILE     65,119
                     JOURN
SALEREC       5      RECORD,( 8 SUB-MEMBERS),      4, 17,144
                     IN FILE TRANS
```

```
SALES#          42        FIELD, NUMERIC IN FILE         40, 41
                          SSLIP
SALES#          63        FIELD, NUMERIC IN FILE         61, 62
                          JOURN
SALES#          82        FIELD, NUMERIC IN FILE         80, 81
                          EXCEPT
SALES#          94        INTERIM NAME                   147
SALETERM        16        TERMINAL NAME                   4
SALPRICE        24        FIELD, NUMERIC IN FILE         21, 95,120,126
                          INVEN
SALPRICE        50        FIELD, NUMERIC IN FILE         47,122
                          SSLIP
SALPRICE        68        FIELD, NUMERIC IN FILE         65,128
                          JOURN
SLIPDATE       150        ASSERTION
SLIPDESC       125        ASSERTION
SLIPREC         41        REPORT_ENTRY,(10               40
                          SUB-MEMBERS),IN FILE
                          SSLIP
SLIPRICE       122        ASSERTION
SLIPTERM        57        TERMINAL NAME                  40
SSITEM          47        GROUP,( 5 SUB-MEMBERS),        41, 60
                          IN FILE SSLIP
SSLIP           40        REPORT,TARGET,                  3, 97, 98,100,101,103,104,106,122,125,132,141,
                          SORTED/KEYED                   153
STOCK#          13        FIELD, CHARACTER IN FILE       12, 91
                          TRANS
STOCK#          22        FIELD, CHARACTER IN FILE       20, 21
                          INVEN
STOCK#          48        FIELD, CHARACTER IN FILE       47
                          SSLIP
STOCK#          66        FIELD, CHARACTER IN FILE       65
                          JOURN
SUBSET                    RESERVED WORD                  153,156
SUBST#          26        FIELD, CHARACTER IN FILE       21, 31,113
                          INVEN
SUBSTIT         26        CHOICE NAME                    112,113,117,129,141
SUBTOT          54        FIELD, NUMERIC IN FILE         41,100,101,104
                          SSLIP
SUBTOT          72        FIELD, NUMERIC IN FILE         62
                          JOURN
TARSET1        153        ASSERTION
TARSET2        156        ASSERTION
TAX             55        FIELD, NUMERIC IN FILE         41,103,104
                          SSLIP
TAX             73        FIELD, NUMERIC IN FILE         62
                          JOURN
TAXCODE         11        FIELD, CHARACTER IN FILE        5,101
                          TRANS
TERM#            6        FIELD, CHARACTER IN FILE        5
                          TRANS
TOTCHRG         56        FIELD, NUMERIC IN FILE         41,106,132,141
                          SSLIP
TOTCHRG         74        FIELD, NUMERIC IN FILE         62
                          JOURN
TR_CUST         90        ASSERTION
TR_INV          93        ASSERTION
TRANS            4        FILE,SOURCE,UNSORTED            2, 88, 91, 95,107,110,138
TRITEM          12        GROUP,( 2 SUB-MEMBERS),         5, 19, 58, 77
                          IN FILE TRANS
```

```
TRSALE        109           ASSERTION
TRSUB         112           ASSERTION
TRYREPL       116           ASSERTION
UPDQUANT      140           ASSERTION
```

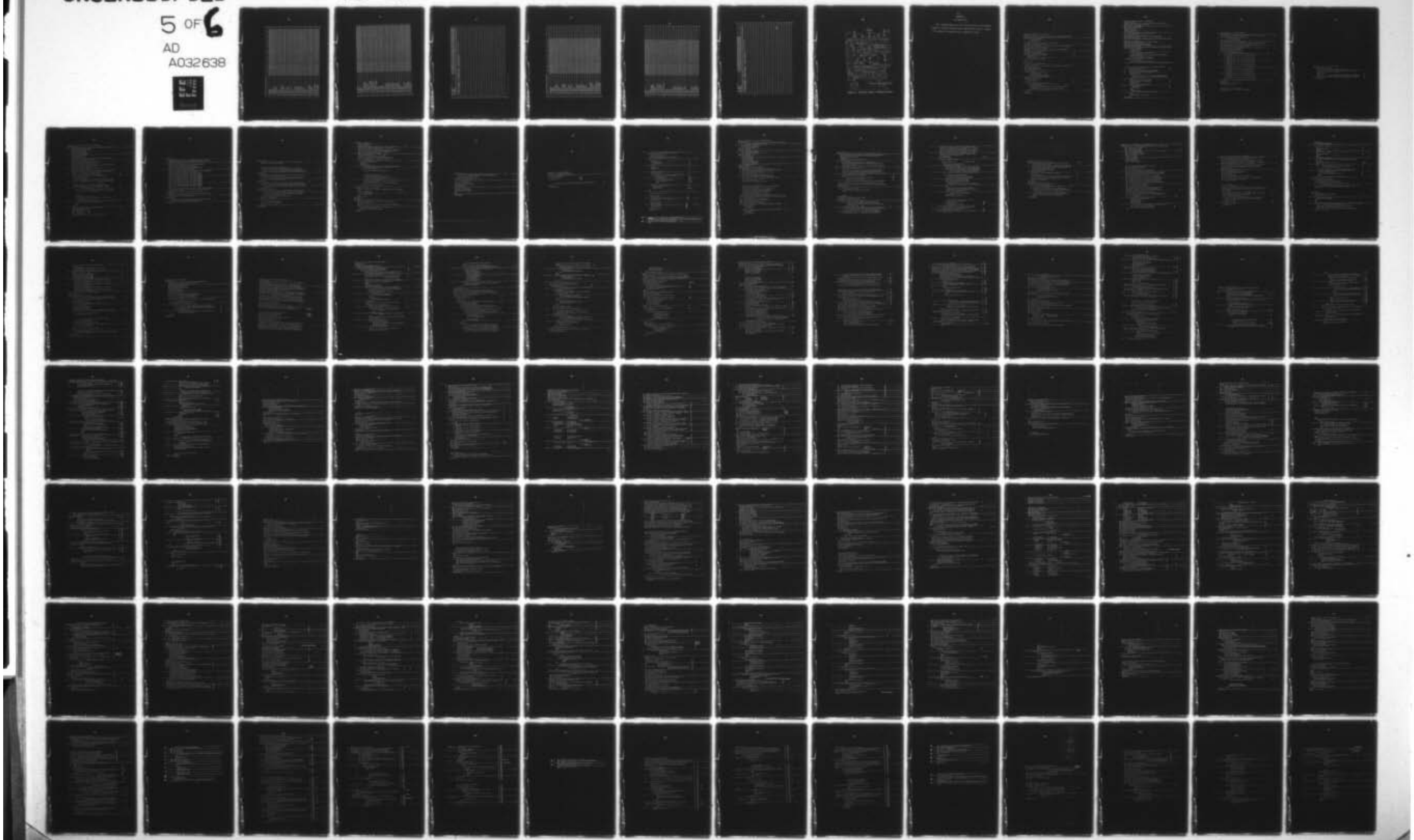
```
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "SSLIP.SALES#=INTERIM.SALES#;"
WARNING(APPARENT AMBIGUITY): FOLLOWING ASSERTION ASSUMED: "SSLIP.CUST#=TRANS.CUST#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "SSLIP.NAME=OLD.CUST.NAME;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "SSLIP.ADDRESS=OLD.CUST.ADDRESS;"
WARNING(APPARENT AMBIGUITY): FOLLOWING ASSERTION ASSUMED: "SSLIP.STOCK#=TRANS.STOCK#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "SSLIP.QUANTITY=TRANS.QUANTITY;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "SSLIP.ENDITEMS=TRANS.ENDITEMS;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.SALES#=INTERIM.SALES#;"
WARNING(APPARENT AMBIGUITY): FOLLOWING ASSERTION ASSUMED: "JOURN.CUST#=TRANS.CUST#;"
WARNING(APPARENT AMBIGUITY): FOLLOWING ASSERTION ASSUMED: "JOURN.STOCK#=TRANS.STOCK#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.QUANTITY=TRANS.QUANTITY;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.EXTEN=SSLIP.EXTEN;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.ENDITEMS=TRANS.ENDITEMS;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.SUBTOT=SSLIP.SUBTOT;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.TAX=SSLIP.TAX;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "JOURN.TOTCHRG=SSLIP.TOTCHRG;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "EXCEPT.SALES#=INTERIM.SALES#;"
WARNING(APPARENT AMBIGUITY): FOLLOWING ASSERTION ASSUMED: "EXCEPT.CUST#=TRANS.CUST#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "EXCEPT.BALANCE=OLD.CUST.BALANCE;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "EXCEPT.CREDLIM=OLD.CUST.CREDLIM;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "EXCEPT.CREDCODE=JOURN.CREDCODE;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.CUST.CUST#=OLD.CUST.CUST#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.CUST.NAME=OLD.CUST.NAME;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.CUST.ADDRESS=OLD.CUST.ADDRESS;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.CUST.CREDLIM=OLD.CUST.CREDLIM;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.INVEN.STOCK#=OLD.INVEN.STOCK#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.INVEN.ITEMDESC=OLD.INVEN.ITEMDESC;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.INVEN.SALPRICE=OLD.INVEN.SALPRICE;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.INVEN.SUBST#=OLD.INVEN.SUBST#;"
WARNING(APPARENT INCOMPLETENESS): FOLLOWING ASSERTION ASSUMED: "NEW.INVEN.ENDSUBST=OLD.INVEN.ENDSUBST;"
```

```
ADJACENCY MATRIX OF NAME RELATIONSHIPS

                                      5        10        15        20        25        30        35        40        45
  1  ADJ_BALC                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2  CALC_EXT                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3  CALC_TAX                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4  CALCCHRG                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5  CALCSAL#                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6  CALCSTOT                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  7  CHOICE.EMPTY             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  8  CHOICE.EXCRLIM           0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  9  CHOICE.SALE              3 0 3 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 10  CHOICE.SUBSTIT           3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 11  CUSTDISK                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 12  DEPSALE                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 13  ERRCHI                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 14  EXCEPT                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 15  EXCEPT.BALANCE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 16  EXCEPT.CREDCODE          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 17  EXCEPT.CREDLIM           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 18  EXCEPT.CUST#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 19  EXCEPT.SALES#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 20  EXCHEC                   0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 21  EXCTERM                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 22  EXIST.JOURITEM           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 23  EXIST.SSITEM             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 24  EXIST.SUBST#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 25  EXIST.TRITEM             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 26  EXLIM                    0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 27  INTERIM.SALES#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0
 28  INVDISK                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 29  JOURDISK                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 30  JOURN                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 31  JOURN.CREDCODE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 32  JOURN.CUST#              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 33  JOURN.ENDITEMS           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 34  JOURN.EXTEN              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
 35  JOURN.JOURITEM           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 36  JOURN.QUANTITY           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
 37  JOURN.SALECODE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
 38  JOURN.SALES#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 39  JOURN.SALPRICE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
 40  JOURN.STOCK#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0
 41  JOURN.SUBTOT             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 42  JOURN.TAX                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 43  JOURN.TOTCHRG            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
 44  JOURREC                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 45  JRPRICE                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0
 46  NEW.CUST                 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 47  NEW.CUST.ADDRESS         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 48  NEW.CUST.BALANCE         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 49  NEW.CUST.CREDLIM         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 50  NEW.CUST.CUST#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 51  NEW.CUST.NAME            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 52  NEW.CUSTREC              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2
 53  NEW.INVEN                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 54  NEW.INVEN.ENDSUBST       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 55  NEW.INVEN.ITEMDESC       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 56  NEW.INVEN.QOH            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 57  NEW.INVEN.SALPRICE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 58  NEW.INVEN.STOCK#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 59  NEW.INVEN.SUBST#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 60  NEW.INVREC             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 61  NJR                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 62  NOREPL                 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 63  NSS                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 64  NSUBST                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 65  NIR                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 66  OLD.CUST               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 67  OLD.CUST.ADDRESS       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 68  OLD.CUST.BALANCE       3 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 69  OLD.CUST.CREDLIM       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 70  OLD.CUST.CUST#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 71  OLD.CUST.NAME          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 72  OLD.CUSTREC            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 73  OLD.INVEN              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 74  OLD.INVEN.ENDSUBST     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 75  OLD.INVEN.ITEMDESC     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 76  OLD.INVEN.CCH          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 77  OLD.INVEN.SALPRICE     0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
 78  OLD.INVEN.STOCK#       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 79  OLD.INVEN.SUBST#       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 80  OLD.INVREC             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 81  POINTER.OLD.CUSTREC    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 82  POINTER.OLD.INVREC     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 83  REORDER                0   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0
 84  SALEREC                0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 85  SALETERM               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 86  SLIPDATE               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 87  SLIPDESC               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 88  SLIPREC                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 89  SLIPPRICE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 90  SLIPTERM               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 91  SSLIP                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 92  SSLIP.ADDRESS          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 93  SSLIP.CUST#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 94  SSLIP.DATEPC           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 95  SSLIP.ENDITEMS         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 96  SSLIP.EXTEN            0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0
 97  SSLIP.ITEMDESC         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 98  SSLIP.NAME             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 99  SSLIP.QUANTITY         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
100  SSLIP.SALES$           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
101  SSLIP.SALPRICE         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
102  SSLIP.SSITEM           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
103  SSLIP.STOCK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
104  SSLIP.SUBTOT           0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0
105  SSLIP.TAX              0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0
106  SSLIP.TOTCHRG          3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0
107  SUBSET.EXCEPT          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
108  SUBSET.SSLIP           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
109  TARSET1                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
110  TARSET2                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
111  TR_CUST                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
112  TR_INV                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
113  TRANS                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
114  TRANS.ACTCODE          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
115  TRANS.CLERK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
116  TRANS.CUST#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0
117  TRANS.DEPT#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
118  TRANS.ENDITEMS         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0
119  TRANS.QUANTITY         0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0
120  TRANS.STOCK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0
121  TRANS.TAXCODE          0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
122  TRANS.TERM#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
123  TRANS.TRITEM   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
124  TRSALE         0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
125  TRSUB          0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
126  TRYREPL        0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
127  UPDQUANT       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A NON-ZERO ENTRY IN ROW I & COLUMN J INDICATES THAT ITEM I PRECEDES ITEM J. THE CODE REPRESENTS THE FOLLOWING TYPE OF RELATIONSHIP:

1 = HIERARCHICAL(SOURCE);  2 = HIERARCHICAL(TARGET);  3 = EXPLICIT DEPENDENCY;  4 = IMPLICIT DEPENDENCY;
5 = POINTING RELATIONSHIP;  6 = STORAGE RELATIONSHIP; 7 = CONDITIONAL DEPENDENCY
```

AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416
UNCLASSIFIED 76-05 NL

5 OF 6
AD
A032638



```
ADJACENCY MATRIX OF NAME RELATIONSHIPS

                                 50        55        60        65        70        75        80        85        90

  1  ADJ_BALC             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2  CALC_EXT             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3  CALC_TAX             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4  CALCCHRG             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5  CALCSAL#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6  CALCSTOT             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  7  CHOICE.EMPTY         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  8  CHOICE.EXCRLIM       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  9  CHOICE.SALE          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 10  CHOICE.SUBSTIT       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 11  CUSTDISK             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 12  DEPSALE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 13  EMPCRI               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 14  EXCEPT               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 15  EXCEPT.BALANCE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 16  EXCEPT.CREDCODE      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 17  EXCEPT.CREDLIM       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 18  EXCEPT.CUST#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 19  EXCEPT.SALES#        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 20  EXCREC               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 21  EXCTERM              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 22  EXIST.JOURITEM       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 23  EXIST.SSITEM         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 24  EXIST.SUBST#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 25  EXIST.TRITEM         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 26  EXLIM                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 27  INTERIM.SALES#       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 28  INVDISK              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 29  JOURDISK             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 30  JOURN                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 31  JOURN.CREDCODE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 32  JOURN.CUST#          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 33  JOURN.ENDITEMS       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 34  JOURN.EXTEN          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 35  JOURN.JOURITEM       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 36  JOURN.QUANTITY       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 37  JOURN.SALECODE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 38  JOURN.SALES#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 39  JOURN.SALPRICE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 40  JOURN.STOCK#         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 41  JOURN.SUBTOT         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 42  JOURN.TAX            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 43  JOURN.TOTCHRG        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 44  JOURREC              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 45  JRPRICE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 46  NEW.CUST             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 47  NEW.CUST.ADDRESS     0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 48  NEW.CUST.BALANCE     0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 49  NEW.CUST.CREDLIM     0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 50  NEW.CUST.CUST#       0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 51  NEW.CUST.NAME        0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 52  NEW.CUSTREC          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 53  NEW.INVEN            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 54  NEW.INVEN.ENDSUBST   0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 55  NEW.INVEN.ITEMDESC   0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 56  NEW.INVEN.COH        0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 57  NEW.INVEN.SALPRICE   0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 58  NEW.INVEN.STOCK#     0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 59  NEW.INVEN.SUBST#     0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 60  NEW.INVREC             0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 61  NJR                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 62  NUREPL                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 63  NSS                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 64  NSUBST                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 65  NTR                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 66  OLD.CUST               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 67  OLD.CUST.ADDRESS       4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4
 68  OLD.CUST.BALANCE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 69  OLD.CUST.CREDLIM       0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 70  OLD.CUST.CUST#         0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 71  OLD.CUST.NAME          0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 72  OLD.CUSTREC            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 73  OLD.INVEN              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 74  OLD.INVEN.ENDSUBST     0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 75  OLD.INVEN.ITEMDESC     0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0
 76  OLD.INVEN.QOH          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 77  OLD.INVEN.SALPRICE     0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0
 78  OLD.INVEN.STOCK#       0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 79  OLD.INVEN.SUBST#       0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 80  OLD.INVREC             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 81  POINTER.OLD.CUSTREC    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 82  POINTER.OLD.INVREC     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0
 83  REORDER                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 84  SALEREC                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 85  SALEITEM               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 86  SLIPDATE               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 87  SLIPDESC               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 88  SLIPREC                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0
 89  SLIPRICE               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 90  SLIPTERM               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 91  SSLIP                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0
 92  SSLIP.ADDRESS          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
 93  SSLIP.CUST#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
 94  SSLIP.DATEPC           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
 95  SSLIP.ENDITEMS         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
 96  SSLIP.EXTEN            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 97  SSLIP.ITEMDESC         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 98  SSLIP.NAME             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
 99  SSLIP.QUANTITY         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
100  SSLIP.SALES#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
101  SSLIP.SALPRICE         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
102  SSLIP.SSITEM           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
103  SSLIP.STOCK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
104  SSLIP.SUBTOT           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
105  SSLIP.TAX              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
106  SSLIP.TOTCHRG          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0
107  SUBSET.EXCEPT          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
108  SUBSET.SSLIP           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0
109  TAXSET1                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
110  TAXSET2                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
111  TR_CUST                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0
112  TR_INV                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0
113  TRANS                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 6 0 0 0 0 0 0 0
114  TRANS.ACTCODE          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
115  TRANS.CLERK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
116  TRANS.CUST#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
117  TRANS.DEPT#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
118  TRANS.ENDITEMS         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
119  TRANS.QUANTITY         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
120  TRANS.STOCK#           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
121  TRANS.TAXCODE          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
122  TRANS.TERM#            0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
    123  TRANS.TRLIEM                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    124  TRSALE                       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    125  TRSUB                        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    126  TRYREPL                      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    127  UPDQUANT                     0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A NON-ZERO ENTRY IN ROW I & COLUMN J INDICATES THAT ITEM I PRECEDES ITEM J. THE CODE REPRESENTS THE FOLLOWING TYPE OF RELATIONSHIP:

1 = HIERARCHICAL(SOURCE);  2 = HIERARCHICAL(TARGET);  3 = EXPLICIT DEPENDENCY;  4 = IMPLICIT DEPENDENCY;
5 = POINTING RELATIONSHIP;  6 = STORAGE RELATIONSHIP; 7 = CONDITIONAL DEPENDENCY
```

ADJACENCY MATRIX OF NAME RELATIONSHIPS

ADJACENCY MATRIX OF NAME RELATIONSHIPS

		93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
1	ADJ_BALC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	CALC_EXT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	CALC_TAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	CALCCHRG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	CALCSAL#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	CALCSTOT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	CHOICE.EMPTY	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	CHOICE.EXCHLIM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	CHOICE.SALE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	CHOICE.SUBSTIT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	CUSTDISK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	DEPSALE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	ERROR1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	EXCEPT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	EXCEPT.BALANCE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	EXCEPT.CRECCODE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	EXCEPT.CREDLIM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	EXCEPT.CUST#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	EXCEPT.SALES#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	EXCHLC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
21	EXCTERM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	EXIST.JOURITEM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	EXIST.SSITEM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	EXIST.SUBST#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	EXIST.TRITEM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	EXLIM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	INTERIM.SALES#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	INVDISK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	JOURDISK	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	JOURN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	JOURN.CHECCODE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	JOURN.CUST#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
33	JOURN.ENDITEMS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
34	JOURN.EXTEN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
35	JOURN.JOURITEM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
36	JOURN.QUANTITY	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
37	JOURN.SALECODE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
38	JOURN.SALES#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
39	JOURN.SALPRICE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
40	JOURN.STOCK#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
41	JOURN.SUBTOT	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
42	JOURN.TAX	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
43	JOURN.TOTCHRG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
44	JOURREC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
45	JRPRICE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
46	NEW.CUST	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
47	NEW.CUST.ADDRESS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
48	NEW.CUST.BALANCE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
49	NEW.CUST.CREDLIM	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
50	NEW.CUST.CUST#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
51	NEW.CUST.NAME	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
52	NEW.CUSTREC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
53	NEW.INVEN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
54	NEW.INVEN.ENDSUBST	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
55	NEW.INVEN.ITEMDESC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
56	NEW.INVEN.QOH	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
57	NEW.INVEN.SALPRICE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
58	NEW.INVEN.STOCK#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
59	NEW.INVEN.SUBST#	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

```
 60  NEW.INVREC              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 61  NJA                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 62  NUMEPL                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 63  NSS                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 64  NSUBST                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 65  NTA                     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 66  OLD.CUST                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 67  OLD.CUST.ADDRESS        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 68  OLD.CUST.BALANCE        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 69  OLD.CUST.CREDLIM        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 70  OLD.CUST.CLST#          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 71  OLD.CUST.NAME           0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 72  OLD.CUSTREC             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 73  OLD.INVEN               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 74  OLD.INVEN.ENDSUBST      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 75  OLD.INVEN.ITEMDESC      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 76  OLD.INVEN.COH           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 0 3
 77  OLD.INVEN.SALPRICE      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 78  OLD.INVEN.STOCK#        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 79  OLD.INVEN.SUBST#        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0
 80  OLD.INVREC              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 81  POINTER.OLD.CUSTREC     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 82  POINTER.OLD.INVREC      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 83  RECNUER                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 84  SALEREC                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0
 85  SALETERM                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 86  SLIPDATE                0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 87  SLIPDESC                0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 88  SLIPREC                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 89  SLIPRICE                0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 90  SLIPTERM                0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 91  SSLIP                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 92  SSLIP.ADDRESS           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 93  SSLIP.CUST#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 94  SSLIP.DATEPC            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 95  SSLIP.ENDITEMS          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 96  SSLIP.EXTEN             0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 97  SSLIP.ITEMDESC          0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 98  SSLIP.NAME              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 99  SSLIP.QUANTITY          0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
100  SSLIP.SALES#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
101  SSLIP.SALPRICE          0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
102  SSLIP.SSITEM            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
103  SSLIP.STOCK#            0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
104  SSLIP.SUBTOT            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
105  SSLIP.TAX               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
106  SSLIP.TOTCHRG           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
107  SUBSET.EXCEPT           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
108  SUBSET.SSLIP            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
109  TARSET1                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
110  TARSET2                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
111  TR_CUST                 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
112  TR_INV                  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
113  TRANS                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
114  TRANS.ACTCODE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
115  TRANS.CLERK#            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
116  TRANS.CUST#             4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
117  TRANS.DEPT#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
118  TRANS.ENDITEMS          0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
119  TRANS.QUANTITY          0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 3 0 3
120  TRANS.STOCK#            0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
121  TRANS.TAXCODE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
122  TRANS.TERM#             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
123  TRAPS.TRITEP                   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
124  TRSALE                         0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
125  TRSUB                          0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
126  TRYMEPL                        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
127  UPDQUANT                       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A NON-ZERO ENTRY IN ROW I & COLUMN J INDICATES THAT ITEM I PRECEDES ITEM J. THE CODE REPRESENTS THE FOLLOWING TYPE OF RELATIONSHIP:

1 = HIERARCHICAL(SOURCE);  2 = HIERARCHICAL(TARGET);  3 = EXPLICIT DEPENDENCY;  4 = IMPLICIT DEPENDENCY;
5 = POINTING RELATIONSHIP;  6 = STORAGE RELATIONSHIP; 7 = CONDITIONAL DEPENDENCY
```

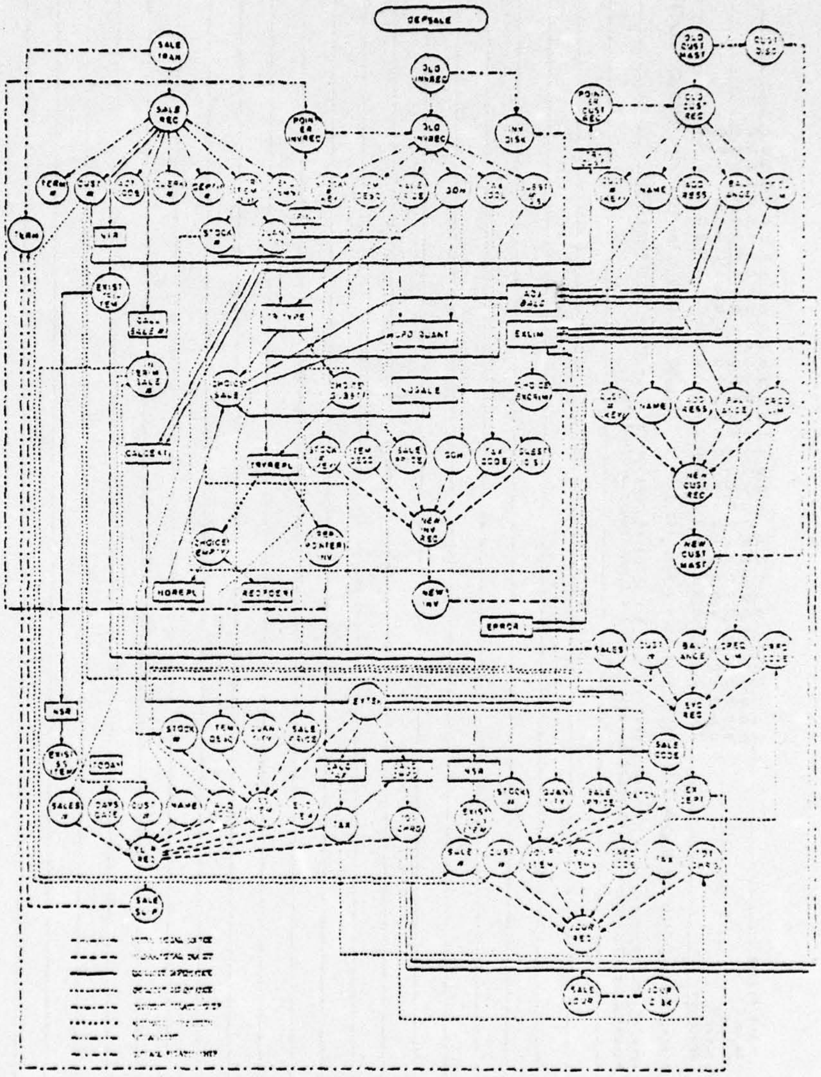


Figure A4 Directed Graph of DEPSALE Problem

APPENDIX B

PL/1 SOURCE CODE

This appendix presents the PL/1 source listings of the important modules of the MODEL Processor whose algorithms were given in Chapter 4. The programs are presented here in alphabetical order.

```
*PROCESS('NST,SM=(2,72,1),N=CHECEND');
 CHECEND: PROC(J);
-/* THIS PROCEDURE LOOKS AT THE NEXT LOCATION IN THE "ENDTAB"(END_TABLE)
 TO SEE WHETHER AN "END" STATEMENT NEEDS TO FOLLOW THIS NODE*/
0DCL 1 FLOWTAB_END BASED(P),
          2 NODE# FIXED BIN,
          2 TYPE CHAR(4);
 DCL J FIXED BIN;
      /* J IS THE CURRENT INDEX TO THE ORDER VECTOR*/
 DCL ORDER(*) FIXED BIN EXT CTL;
 DCL #LOOPS FIXED BIN EXT;
 /*NUMBER OF LOOPS(DETECTED BY 'FLOWCPT) ENTERED IN DO AND END TABLES*/
 DCL #END FIXED BIN STATIC INIT(1);
 DCL ENDTAB(*) FIXED BIN CTL EXT;
 /*TABLE OF INDICES TO ORDER VECTOR AFTER WHICH  "END" STATEMENT TO
 APPEAR */
-IF #END> #LOOPS THEN RETURN;                                          1
-DO WHILE(ENDTAB(#END)=J);
 /*GENERATE "END" FLOWCHART ENTRY*/                                    2
 LOCATE FLOWTAB_END FILE(FLOWTAB);
      NODE#=ORDER(J);                                                  3
      TYPE='END';
      #END=#END + 1;
      IF #END>#LOOPS THEN RETURN;
0END;
-END CHECEND;

*PROCESS('NST,MACRO,N=CHECLAB');
 CHECLAB: PROC(NODE);
 /* THIS PROC CHECKS IF ANY LABEL NEEDS TO BE GENERATED AT THIS POINT */
 DCL NODE# FIXED BIN;
 %DCL MAX#_LABELS FIXED;
 %DCL MAX_LEN_LABEL FIXED;
 %MAX_LEN_LABEL=14;
 %MAX#_LABELS=10;
      DCL 1 FLOWTAB_LAB BASED (P),
            2 NODE# FIXED BIN,
            2 TYPE CHAR(4),       /*LAB*/
            2 LABEL CHAR(MAX_LEN_LABEL);
 DCL LABEL(MAX#_LABELS) FIXED BIN EXT;
 DCL #LABELS FIXED BIN EXT;
 DO I=1 TO #LABELS;                                                 1
 IF NODE =LABEL(I) THEN
      DO;                                                           2
 /* GENERATE LABEL*/
            LOCATE FLOWTAB_LAB FILE(FLOWTAB);
            FLOWTAB_LAB.NODE#=0;
            FLOWTAB_LAB.TYPE='LAB';
            PUT STRING(FLOWTAB_LAB.LABEL) EDIT('$L',NODE) (A,P'999');
            LABEL(I)=0;
              /*SO THAT LABEL WILL NEVER BE GENERATED TWICE*/
      RETURN;
      END;
 END;
-END CHECLAB;
```

```
 *PROCESS('NST,MACRO,SM=(2,72,1),N=CHECKDO');
  CHECKDO: PROC(J);
 -/*THIS PROCEDURE LOOKS AT THE NEXT LOCATION IN DO_TABLE TO SEE IF
  CURRENT NODES NEED A "DO" STATEMENT*/
  %DCL MAX_LEN_EXIST FIXED;
  %DCL MAX_LEN_FOREACH FIXED;
  %DCL MAX_LEN_NAME FIXED;
  %MAX_LEN_EXIST=32;
  %MAX_LEN_NAME=10;
  %MAX_LEN_FOREACH=18;
  DCL J FIXED BIN;
  /*J IS THE CURRENT INDEX TO THE ORDER VECTOR*/
  /*INDEX TO DO TABLE*/
 -DCL #DO FIXED BIN STATIC INIT(1);
 0DCL UPBOUND ENTRY(CHAR(*),FIXED BIN) RETURNS(FIXED BIN);
 0DCL(UPB,#SUBS)FIXED BIN;
  /*FUNCTION WHICH RETURNS THE UPPER BOUND OF A REPEATING GROUP OR FIELD
  /*
  /*UPB:SET TO UPBOUND*/
  /*#SUBS: NUMBER OF SUBSCRIPTS OF REPEATING GROUP*/
 0DCL ORDER(*) FIXED BIN EXT CTL;
 0DCL REPEATING_GROUP CHAR(MAX_LEN_NAME);
  /*NAME OF GROUP OF FIELD THAT REPEATS*/
 0DCL #LOOPS FIXED BIN EXT;
  /*NUMBER OF LOOPS DETECTED BY 'FLOWCPT' ENTERED IN DO AND END TABLES*/
  DCL 1 FLOWTAB_DO BASED(P),
            2 NODE# FIXED BIN,
            2 TYPE CHAR(4),
            /*TYPE'DO  '*/
            2 FOREACH_NAME CHAR(MAX_LEN_FOREACH),
            2 UPPER_TYPE CHAR(1),
            /*F=FIXED(CONSTANT) UPPER BOUND,V=VARYING UPPER BOUND*/
            2 UPPER# FIXED DEC,
            2 UPPER_NAME CHAR(MAX_LEN_EXIST);
  /*FLOWCHART TABLE ENTRY FOR DO STATEMENT*/
       DCL 1 DOTAB(*) EXT CTL,
            2 LOC FIXED BIN,
            2 DO_LOOP_VAR CHAR(MAX_LEN_FOREACH);
  /*DOTAB FILLED BY FLOWCPT;LOC:INDEX TO ORDER VECTOR GIVING LOCATIONS OF


  "DO"; DO_LOOP_VAR:"FOREACH" VARIABLE THAT GOVERNS ITERATION*/
 -     IF #DO> #LOOPS THEN RETURN;                                    1
 -     DO WHILE(LOC(#DO)=J);                                          2
 0        REPEATING_GROUP=SUBSTR(DO_LOOP_VAR(#DO),9);
 0        /*FIND BOUND OF REPEATING GROUP OR FIELD*/
          UPB=UPBOUND(REPEATING_GROUP,#SUBS);
 0        IF UPB=0|#SUBS=0
          THEN DO;
             CALL PNETERR('(INCONSISTENCY):"'||DO_LOOP_VAR(#DO)||
             '" DOES NOT CORRESPOND TO A REPEATING GROUP OR FIELD');
             RETURN;
          END;
 0        /*GENERATE "DO" FLOWCHART ENTRY*/
 0        LOCATE FLOWTAB_DO FILE(FLOWTAB);                            3
          NODE#=ORDER(J);                                             4
          TYPE='DO';
          FOREACH_NAME=DO_LOOP_VAR(#DO);
 0        IF #SUBS=1 THEN DO;                                         5
             UPPER_TYPE='F';
             UPPER#=UPB;                                              6
          END;
 0        ELSE DO;
             UPPER_TYPE='V';
             UPPER_NAME='EXIST.'||REPEATING_GROUP;
          END;
 -        #DO=#DO + 1;
          IF #DO> #LOOPS THEN RETURN;
 0     END;
 -END CHECKDO;
```

```
*PROCESS('MACRO,NST,EXTREF,N=CODEGEN');
 CODEGEN:PROC;
 /* THIS PROCEDURE USES THE FLOWCHART RECORDS TO CALL THE ROUTINES
 WHICH DO THE CODE GENERATION. */
 DCL 1 FLOWTAB_ENTRY_PROTO BASED(FLOW_PTR),
          2 NODE# FIXED BIN,
          2 NODE_TYPE CHAR(4),
 UNTIL_EOF BIT(1) INIT('1'B);
 ON ENDFILE(FLOWTAB) GO TO FINISH_UP;
0/* WE PICK UP THE RECORDS FROM THE FILE.  THE ON ENDFILE WILL TELL
 US WHEN WE ARE DONE.  THUS THE INFINITE LOOP. */
 /* OPEN THE INPUT FILE WHICH WILL BE USED BY THE SUBROUTINES. */
 OPEN FILE(FLOWTAB) INPUT RECORD SEQUENTIAL;
0/*OPEN ALL THE OUTPUT PL/1 FILES*/
 OPEN FILE(PL1ON)OUTPUT RECORD SEQL,                                1
 FILE(PL1EX) OUTPUT RECORD SEQL,
 FILE(PL1PROC)OUTPUT RECORD SEQL;
-LOOP: DO WHILE(UNTIL_EOF);
     READ FILE(FLOWTAB) SET(FLOW_PTR);                              2
     IF NODE_TYPE='ASSN' THEN CALL CPRCCCD(FLOW_PTR);
         ELSE DO:                                                   3
     IF NODE_TYPE='RECD' THEN CALL GENICCD(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='FLDI' THEN CALL GIMFLD(FLOW_PTR);
         ELSE DO:
     IF NODE_TYPE='INTI' THEN CALL GIMFLD(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='PRTN' THEN CALL GENICCD(FLOW_PTR);
         ELSE DO:
     IF NODE_TYPE='MODL' THEN CALL GMCDCD(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='GOTO' THEN CALL GENGOTO(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE= 'LAB' THEN CALL GENLAB(FLOW_PTR);
         ELSE DO:
     IF NODE_TYPE='END' THEN CALL GENEND(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='DO' THEN CALL  GENDO(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='PSET' THEN CALL GENPSET(FLOW_PTR);
         ELSE DO;
     IF NODE_TYPE='COND' THEN CALL GENCOND(FLOW_PTR);
      ELSE DO;
     IF NODE_TYPE='TEXT' THEN CALL GENTEXT(FLOW_PTR);
 END LOOP;                                                          4
0FINISH_UP:
0/* CLOSE OUT THE INPUT FILE. */
 CLOSE FILE(FLOWTAB);


 /*CLOSE OUT THE OUTPUT FILES*/
 CLOSE FILE(PL1ON), FILE(PL1EX),FILE(PL1PROC);
 END CODEGEN;
```

```
*PROCESS('NST,MACRO,N=CRADJMT,EXTREF');
 CRADJMT: PROC;
      %INCLUDE INCLIB(DDICT);
      DCL ADJMAT(DICTINC,DICTINC) FIXED BIN EXT CTL;
       /* ADJMAT IS THE ADJACENCY MATRIX OF USER-NAME RELATIONSHIPS */
      /* ALLOCATE AND INITIALIZE ADJMAT */
       ALLOCATE ADJMAT;
      ADJMAT=0;                                                                    2
      CALL ENDPREL;  /* ENTER DEPENDENCY RELATIONSHIPS INTO ADJMAT */             3
      CALL ENHPREL;  /* ENTER HIERARCHICAL RELATIONSHIPS INTO ADJMAT */           4
      CALL ENPTREL;  /* ENTER POINTER-FILE RELATIONSHIPS INTO ADJMAT */           5
 END  CRADJMT;
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=CRDICT,EXTREF');
 CRDICT: PROC;
     /*THIS PROCEDURE CREATES A DICTIONARY CORRESPONDING A NUMBER TO
         EACH FULLY QUALIFIED NAME*/
      % INCLUDE INCLIB (DSEDIR);
     %INCLUDE INCLIB (DANY);
      %INCLUDE INCLIB(DASSNM);
      %DCL MAX#_RETPTRS FIXED;
      %DCL MAX_LEN_NAME FIXED;
         %MAX#_RETPTRS=200;
      %MAX_LEN_NAME=10;
      %INCLUDE INCLIB(DDICT);
      DCL CHPTELP ENTRY (CHAR(*)) RETURNS (CHAR(LEN_DICT_ENTRY)VAR);
      DCL PRINT_LINE CHAR(120) BASED(PRL);
      DCL PRL POINTER STATIC;
      DCL STMT_TYPE_ABBREV CHAR(4) STATIC;
      DCL CURRENT_NAME_PTR(MAX#_RETPTRS) POINTER;
      DCL DIR_KEY (#DIREN) POINTER CTL EXT;
      /*TABLE OF POINTERS TO DIRECTORY ENTRIES IN ALPHABETICAL ORDER;
        ALREADY ALLOCATED AND SET BY 'XREF'*/
      DCL  #DIREN     FIXED BIN EXT;
      /*'#DIREN' IS NUMBER OF ENTRIES IN DIRECTORY*/
      DCL #CURRENT_NAME_SE FIXED BIN;
      /* NUMBER OF STORAGE ENTRIES WITH CURRENT NAME, SET BY 'RETREVE' */
      DCL START POINTER EXT; /*POINTER TO BEGINNING OF DIRECTORY*/
      DCL CURRENT_NAME CHAR(MAX_LEN_NAME) STATIC;
      DCL RESERVE ENTRY(CHAR(*)) RETURNS(BIT(1));
      DO I=1 TO #DIREN; /* FOR EACH DIRECTORY ENTRY*/                      1-2
        /*SET DIRECTORY POINTER TO NEXT ALPHABETICAL NAME*/
        DIRECTORY_PTR=DIR_KEY(I);
        CURRENT_NAME=KEY_NAME;
        IF /*SYSTEM_ASSIGNED NAME*/ SUBSTR(CURRENT_NAME,1,1)='$' THEN;
         ELSE IF CURRENT_NAME=' ' THEN;  /* BLANK NAMES IGNORED */
         ELSE IF RESERVE(CURRENT_NAME) THEN; /* DONT PROCESS RESERVED
                                              WORDS */
       ELSE
       DO; /*'CURRENT_NAME' IS A USER_DEFINED NAME*/
         /*RETRIEVE ALL STORAGE ENTRIES WITH CURRENT _NAME*/
         CALL RETREVE(CURRENT_NAME,'',START,CURRENT_NAME_PTR,                3
                  #CURRENT_NAME_SE);
        DO J=1 TO #CURRENT_NAME_SE;   /* FOR EACH OF THE RETRIEVED
                                  STORAGE ENTRIES WITH CURRENT_NAME */
          STORAGE_PTR=CURRENT_NAME_PTR(J);



          DP=DATA_PT;
          IF CURRENT_NAME =NAME(1) THEN
          DO; /* THIS IS THE STATEMENT WHERE THE CURRENT NAME IS
                DEFINED */
            STMT_TYPE_ABBREV=ANY_STMT.TYPE;
            CALL RESTMT; /* CALL ROUTINE TO BRANCH ON STMT TYPE AND THEN    4
                 ENTER NAME IN DICT ACCORDINGLY  */
          END;
        END;
      END;
     END;
      /* ENTER SPECIAL NAMES INTO THE DICTIONARY  */                          10
      STMT_TYPE_ABBREV= 'SPCN'     ;
     CALL GETCELP('CHOICE     ');
     CALL GETCELP('EXIST      ');
     CALL GETCELP('LEN        ');
     CALL GETCELP('POINTER    ');
     CALL GETCELP('SUBSET     ');
     CALL ALPDICT;    /* PUT DICT IN ALPHABETICAL ORDER */                    11
```

```
1BRSTMT: PROC;                                                              4
      /*THIS PROCEDURE DETERMINES WHICH PROCEDURE TO CALL TO ENTER THE
      'CURRENT NAME' (DEFINED IN THE CURRENT STORAGE ENTRY OF TYPE
      'STMT_TYPE_ABBREV') AS A FULLY QUALIFIED NAME INTO THE
      DICTIONARY ('DICT') */
      DCL PARENT ENTRY(POINTER) RETURNS(CHAR(MAX_LEN_NAME));
      DCL TEMP_PARENT CHAR(MAX_LEN_NAME) VAR;
      IF STMT_TYPE_ABBREV='FLD ' THEN CALL ENTMEM;
      ELSE IF STMT_TYPE_ABBREV='ASTG' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='ASSR' THEN;
      ELSE IF STMT_TYPE_ABBREV='ASTX' THEN;
      /*'ASSR' AND 'ASTX' NAMES DO NOT GET ENTERED SINCE THE ASSERTION
      NAME HAS ALREADY BEEN ENTERED IN 'ASTG'*/
      ELSE IF STMT_TYPE_ABBREV='GRP ' THEN CALL ENTMEM;
      ELSE IF STMT_TYPE_ABBREV='INTR' THEN CALL ENTINTR;
      ELSE IF STMT_TYPE_ABBREV='FILE' THEN CALL ENTFILE;
      ELSE IF STMT_TYPE_ABBREV='RECD' THEN CALL ENTRECD;
      ELSE IF STMT_TYPE_ABBREV='REPT' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='RPTN' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='MODL' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='DISK' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='PRNT' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='CARD' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='TAPE' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='TERM' THEN CALL ENTSING;
      ELSE IF STMT_TYPE_ABBREV='PNCH' THEN CALL ENTSING;
      ELSE
      /* 'CURRENT_NAME' OF DIRECTORY WAS FOUND AS FIRST NAME IN CURRENT
      STORAGE ENTRY. HOWEVER, THE STORAGE ENTRY STATEMENT TYPE WAS
      FOUND TO BE ILLEGAL*/
      CALL SYSERR (STMT_TYPE_ABBREV||' ILLEGAL STMT TYPE');
1ENTSING: PROC;                                                             5
      /*THIS PROC ENTERS THE SINGLE 'CURRENT_NAME' (WITHOUT TRAILING
      BLANKS) INTO THE DICTIONARY*/
      CALL ENTDICT(CHPTRLB(CURRENT_NAME));
 END ENTSING;
1ENTINTR: PROC;                                                             6
      /* THIS PROC WILL ENTER THE CURRENT_NAME, AN INTERIM NAME, WITH
          THE PREFIX 'INTERIM.' INTO THE DICTIONARY    */
      CALL ENTDICT('INTERIM.'|| CHPTRLB(CURRENT_NAME));
 END ENTINTR;
```

```
1ENTMEM: PROC;
      /*ENTER 'CURRENT_NAME', A GROUP OR FIELD, WITH ITS PARENT FILE            7
      TO QUALIFY IT, IN THE DICTIONARY*/


      TEMP_PARENT=CHRTRLB(PARENT(STORAGE_PTR));
      /* 'ENEWOLD' WILL ENTER THE QUALIFIED NAME WITH BOTH PREFIXES
         'OLD.' AND 'NEW.' ONLY IF THE PARENT FILE NAME IS BOTH AN
         INPUT AND OUTPUT FILE: OTHERWISE IT WILL ENTER THE ONAME AS IS*/
      CALL ENEWOLD(TEMP_PARENT||'.'||CHRTRLB(CURRENT_NAME),TEMP_PARENT);
 END ENTMEM;
1ENTFILE: PROC;                                                                   8
      /* THIS PROC WILL ENTER THE CURRENT_NAME, A FILE NAME, INTO DICT */
      TEMP_PARENT=CHRTRLB(CURRENT_NAME);
      /* 'ENEWOLD' WILL ENTER THE FILE NAME WITH BOTH PREFIXES 'OLD.' &
            'NEW.' ONLY IF THE FILE NAME IS BOTH INPUT & OUTPUT */
      CALL ENEWOLD(TEMP_PARENT,TEMP_PARENT);
 END ENTFILE;
1ENTRECD: PROC;
      /* THIS PROC WILL ENTER THE CURRENT_NAME, A RECD NAME, INTO DICT */         8
      CALL ENEWOLD(CHRTRLB(CURRENT_NAME),CHRTRLB(PARENT(STORAGE_PTR)));
      /* 'ENEWOLD' WILL ENTER   RECORD NAME AND ALSO THE PREFIXES 'OLD.' &
            'NEW.' ONLY IF THE PARENT FILE OF THE RECORD IS BOTH AN INPUT &
            OUTPUT FILE  */
 END ENTRECD;
1ENEWOLD: PROC(ONAME,PAR);
      /* THIS PROC,"ENTER NEW/OLD",  CHECKS WHETHER THE PARENT FILE NAME          9
           'PAR'     IS BOTH AN INPUT AND OUTPUT FILE. IF SO, IT ENTERS
           THE CURRENT QUALIFIED NAME 'ONAME' INTO THE DICTIONARY WITH
           BOTH  THE PREFIX 'NEW.' AND 'OLD.'; OTHERWISE IT ENTERS THE
           ONAME AS IT IS INTO THE DICT   */
      DCL IOFILE ENTRY(CHAR(*)VAR) RETURNS(BIT(1));
      DCL ONAME CHAR(*) VAR;
      DCL PAR     CHAR(*) VAR;
      IF IOFILE(PAR) THEN
      DO;
         CALL ENTDICT('NEW.'||ONAME);
         CALL ENTDICT('OLD.'||ONAME);
      END;
      ELSE CALL ENTDICT(ONAME);
 END ENEWOLD;
-END ARSTMT;
```

```
1GETCELP: PROC(CELP);
     %DCL MAX#_CELP FIXED;
     %MAX#_CELP=50;
     /* THIS PROC ENTERS ALL THE 'LENGTH', 'EXIST', 'CHOICE', AND
     'POINTER' AND 'SUBSET' NAMES INTO THE DICTIONARY */
     DCL TONAME CHAR(LEN_DICT_ENTRY) VAR;
     DCL CELP CHAR (*); /* 'CHOICE', 'EXIST', 'LEN', OR 'POINTER'*/
     DCL CELPCK ENTRY(CHAR(*)VAR,CHAR(*))  RETURNS(BIT(1));
     DCL DICTEX ENTRY (CHAR(*)VAR) RETURNS (BIT(1));
     DCL CELP_PTR (MAX#_CELP) POINTER;
     /*POINTERS TO ENTRIES WITH A 'CELP' NAME*/
     DCL #CELP_PTR FIXED BIN;
     /* NUMBER OF ENTRIES WITH A 'CELP' NAME*/
     /* RETRIEVE ALL STORAGE ENTRIES WITH A 'CELP' NAME*/
     CALL RETREVE (CELP,'',START,CELP_PTR,#CELP_PTR);
     DO I=1 TO #CELP_PTR;
           /* FOR EACH STORAGE ENTRY WITH A 'CELP' NAME */
        STORAGE_PTR=CELP_PTR(I);
        DP=DATA_PT;
        NAMEIND=2;
        DO J=1 TO #NMS;
          /* FOR EACH QUALIFIED NAME IN THE CURRENT STORAGE ENTRY,
             DETERMINE WHETHER IT IS THE 'CELP' NAME */
           IF NAME (NAMEIND)=CELP THEN
           DO;
              TONAME=CHPTRLB(NAME(NAMEIND))||'.';



              IF #COMPONENTS(J)=2 THEN
              TONAME=TONAME||CHPTRLB(NAME(NAMEIND+1));
              ELSE IF #COMPONENTS(J)=3 THEN
               TONAME=TONAME||CHPTRLB(NAME(NAMEIND+1))||'.'||
                CHPTRLB(NAME(NAMEIND+2));
              ELSE CALL PNETERR(TONAME||' ILLEGAL '||CELP||' NAME');
              IF ¬DICTEX(TONAME) THEN
              IF CELPCK(TONAME,CELP)  THEN CALL ENTDICT(TONAME);
           END;
           NAMEIND=NAMEIND+#COMPONENTS(J);
        END;
     END;
0DICTEX: PROC(TONAME) RETURNS(BIT(1));
     /*THIS PROC DETERMINES IF 'TONAME' ALREADY EXISTS IN THE DICTIONARY
     */
     DCL TONAME CHAR(*) VAR;
     DO K=1 TO DICTIND;
        IF DICT (K)= TONAME
        THEN RETURN ('1'B);
     END;
     RETURN ('0'B);
0END DICTEX;
0END GETCELP;
1ENTDICT: PROC (DICTENT);
     /*THIS PROCEDURE ENTERS 'DICTENT' INTO THE DICTIONARY */
     DCL DICTENT CHAR(*)  VAR;
     DICTIND=DICTIND+1;
     IF DICTIND>MAX#_QNAMES THEN CALL SYSERR ('DICT SIZE EXCEEDED');
     ELSE
     DO;   /* ENTER NAME AND TYPE INTO THE DICTIONARY  */
        DICT(DICTIND) = DICTENT;
         DICTYPE(DICTIND)=STMT_TYPE_ABBREV;
      END;
 END ENTDICT;
```

```
1ALPDICT: PROC;
      /* THIS PROC ALPHABETIZES THE DICT USING A BUBBLE SORT */
      /* THIS /* IT ALSO RE/ARRANGES THE DICTYPE ACCORDINGLY */
      %DCL MAX_LEN_DICT_ENTRY  FIXED;
      %MAX_LEN_DICT_ENTRY=32;
  DCL NAME CHAR(MAX_LEN_DICT_ENTRY)  VAR;
  DCL TEMP_DICTYP= CHAR(4);
      DCL (I,J) FIXED BIN;
  DO I=1 TO (DICTINO-1);
  DO J=2 TO (DICTINO-I+1);
  IF DICT(J)<DICT(J-1) THEN DO;
  NAME=DICT(J-1);
  DICT(J-1)=DICT(J);
  DICT(J)=NAME;
  TEMP_DICTYPE=DICTYPE(J-1);
  DICTYPE(J-1)=DICTYPE(J);
  DICTYPE(J)=TEMP_DICTYPE;
  END;
  END;
  END ALPDICT;
 -END CRDICT;
```

```
*PROCESS('AST,N=CRPATHS,EXTREF');
 CRPATHS: PROC(ADJMAT,PATHMAT,N);
     DCL (ADJMAT,PATHMAT)(*,*) BIT(*);
 DCL OUT CHAR(200) VAR;                                  1
      PATHMAT=ADJMAT;
 DO J=1 TO N;                                            2-6
     DO I=1 TO N;                                        3-5


         IF PATHMAT(I,J) THEN PATHMAT(I,*)=PATHMAT(I,*)|PATHMAT(J,*);            4
     END;
 END;
    END CRPATHS;
```

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=CYCLES');
 CYCLES: PROC (A,P,N) ;
      DCL (A,P) (*,*) BIT(*);
     DCL USED (N) BIT(1);
   DCL (ROOT,REACHJ(N),PATH(N+1)) FIXED BIN;
   DCL PATH_EXTENDED BIT(1);

   DO ROOT=1 TO N;                                          1
      DO K=ROOT TO N;
        REACHJ(K)=ROOT;                                     2
        USED(K)='0'B;                                       3
      END;

      LEVEL=1;                                              4
      PATH(1)=ROOT;                                         5
      I=ROOT;                                               6

      DO WHILE(LEVEL¬=0);
      PATH_EXTENDED='0'B;
      JMIN=REACHJ(I);                                       7
      DO J=JMIN TO N WHILE( (LEVEL¬=0)&¬PATH_EXTENDED);     8
        IF A(I,J)&P(J,ROOT)&¬USED(J) THEN DO;               9
           PATH_EXTENDED='1'B;
           CALL EXTEND_PATH;                                18-23
           IF J=ROOT THEN DO;
              CALL RECYCLE(PATH,LEVEL);                     24
              CALL BACKTRACK;                               13-17
              END;
           END;
     END;
      IF ¬PATH_EXTENDED THEN DO;                            12
         REACHJ(I)=ROOT;
         CALL BACKTRACK;                                    13-17
         END;
   END;
   END;

   /* INTERNAL SUBROUTINES */

   BACKTRACK: PROC;
      USED(I)='0'B;                                        13
      LEVEL=LEVEL-1;                                       14
      IF LEVEL¬=0 THEN I=PATH(LEVEL);                      15
    END BACKTRACK;

    EXTEND_PATH: PROC;
      USED(J)='1'B;                                        18
      REACHJ(I)=J+1;                                       19
      LEVEL=LEVEL+1;                                       20
      PATH(LEVEL)=J;                                       21
      I=J;                                                 22
    END EXTEND_PATH;
   END CYCLES;
```

```
DELIM: PROC(STR,DELCHAR) RETURNS(FIXED BIN);
/* RETURNS THE NUMBER OF CHARACTERS UP TO THE SPECIFIED DELIMETER */
DCL STR CHAR(*) VAR, DELCHAR CHAR(*);
RETURN( INDEX(SUBSTR(STR,I),DELCHAR)      -1);
END;
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=ENEXOP,EXTREF');
 ENEXOP: PROC;
      /*THIS PROCEDURE ENTERS ALL THE EXPLICIT DEPENDENCY RELATIONSHIPS
           (GIVEN BY USER ASSERTIONS) IN THE
      ADJACENCY MATRIX 'ADJMAT'*/
      %DCL MAX_LEN_GNM FIXED;
      %DCL  EXPL_DEP_CODE FIXED;
 %DCL AUTO_DEP_CODE FIXED;
 %AUTO_DEP_CODE=3;
      %DCL MAX#_ASSERT FIXED;
      %DCL MAX_LEN_NAME FIXED;
      %DCL MAX#_EMPTY_PTR FIXED;
 %DCL COND_DEP_CODE FIXED;
 %COND_DEP_CODE=7;
      %MAX#_EMPTY_PTR=5;
      %MAX_LEN_GNM=32;
      %EXPL_DEP_CODE=3;
      %MAX#_ASSERT=100;
      %MAX_LEN_NAME=10;
      %INCLUDE INCLIB(CSFDIR);
      %INCLUDE INCLIB(DDICT);
      %INCLUDE INCLIB(DASSNM);
 %DCL MAX#_LABELS FIXED;
 %MAX#_LABELS=10;
 DCL LABEL(MAX#_LABELS) FIXED BIN EXT;
 DCL #LABELS FIXED BIN EXT INIT(0);
      DCL CURRENT_ASSERT_NAME CHAR(MAX_LEN_NAME)VAR;
      DCL CHPTPLB  ENTRY(CHAR(*)) RETURNS(CHAR(MAX_LEN_GNM)VAR);
      DCL GNAME CHAR(MAX_LEN_GNM)VAR;
      DCL ADJMAT(DICTIND,DICTIND)FIXED BIN EXT  CTL;
      DCL ST_IND FIXED BIN;
 DCL COND ENTRY(CHAR(*))RETURNS(BIT(1));
      DCL DICT# ENTRY(CHAR(*)VAR)RETURNS(FIXED BIN);
      DCL VLGNAME ENTRY (FIXED BIN,FIXED BIN,POINTER,
                               FIXED BIN,CHAR(*)VAR,BIT(1));
      DCL TRY_ENTER_IN_MATRIX BIT(1);
      DCL START POINTER EXT;
      DCL ASSERT_PTR(MAX#_ASSERT) POINTER;



      DCL (#ASSR,#ASTC)  FIXED BIN;
      DCL CONSONM ENTRY(FIXED BIN,FIXED BIN,POINTER)RETURNS
      (CHAR(MAX_LEN_GNM)VAR);
      DCL $ASSERT CHAR(MAX_LEN_NAME) INIT('$ASSERT');
      DCL EMPTY CHAR(MAX_LEN_NAME)INIT('EMPTY');
      DCL CHOICE CHAR(MAX_LEN_NAME)INIT('CHOICE');
      DCL EMPTY_PTR(MAX#_EMPTY_PTR) POINTER;
      DCL #EMPTY_PTR FIXED BIN;
      DCL TEMP_TAB_FILE CHAR(MAX_LEN_GNM)VAR;
 DCL P POINTER, #REC_PTR FIXED BIN, REC_PTR(64) POINTER;
 DCL $RECD CHAR(MAX_LEN_NAME) INIT('$RECD'), $PPTN CHAR(MAX_LEN_NAME)
        INIT('$PPTN');
 DCL FILENAME_RETRVE CHAR(MAX_LEN_NAME);
 DCL FILEST ENTRY(CHAR(*)) RETURNS(CHAR(2));
 DCL FILE_ST_TYPE CHAR(2);
 DCL HAPE_FILE_NAME CHAR(MAX_LEN_NAME);
 DCL RECNAME CHAR(MAX_LEN_GNM)VAR;
      CALL RETRVE($ASSERT,       'ASSR',START,ASSERT_PTR,#ASSR);            1
      DO I=1 TO #ASSR;                                                       2
         ST_IND=0;
         CALL GTASNMS;
      END;
      CALL RETRVE($ASSERT,       'ASTC',START,ASSERT_PTR,#ASTC);            1
      DO I=1 TO #ASTC;                                                       2
         ST_IND=1;
         CALL GTASNMS;
      END;
```

```
  IGTASNMS: PROC;
   /* GET ALL THE SOURCE/TARGET NAMES TO THE ASSERTION AND ENTER THE
        EXPLICIT DEPENDENCY RELATIONSHIP BETWEEN ASSERTION AND EACH NAME;
        ONLY EXCEPTION: IF ASSERTION USES THE "REPLACE" FUNCTION  */
        STORAGE_PTR=ASSERT_PTR(I);
        DP=DATA_PT;
        CURRENT_ASSERT_NAME=CHPTRLB(NAME(1));
        NAMEIND=2;
0       DO J=1 TO #NMS;                                                        3
0          QNAME=CONSONM(NAMEIND,#COMPONENTS(J),STORAGE_PTR);
         CALL VLDNAME(NAMEIND,#COMPONENTS(J),STORAGE_PTR,ST_IND,QNAME,
          TRY_ENTER_IN_MATRIX);
C         /*'TRY_ENTER_IN_MATRIX' RETURNS '1' IF THERE SHOULD BE SUCH AN
          ATTEMPT*/
C         IF TRY_ENTER_IN_MATRIX THEN
          DO;
-         /*IF ASSERTION INVOLVES THE "REPLACE" FUNCTION DON'T ENTER ITS
          TARGET IN THE ADJACENCY MATRIX, BECAUSE THAT COULD CAUSE A
          CYCLE: THE "REPLACE" FUNCTION ITSELF WILL PERFORM THE
          REPLACEMENT/STACKING AND BRANCH. DO LOOK FOR THE "CHOICE_EMPTY"
          CONDITION TO MAKE THE CONNECTION*/
0         IF FCN='REPLACE' & ST_IND=1 THEN                                     4
         DO;                                                                   10
             DO I1=1 TO DICTIND WHILE(DICT(I1)<'EMPTZ');
                IF DICT(I1)='CHOICE.EMPTY'                                     11
                THEN                                                           12
                CALL ENDRMAT(DICT(I1),CURRENT_ASSERT_NAME,AUTO_DEP_CODE);
             END;
   /* ENTER NODE NUMBER OF LOCATION WHERE REPLACEMENT SHOULD GO IN THE
   "LABEL" TABLE; THE LABEL TABLE WILL BE CHECKED BY THE CHECLAB ROUTINE*/
   I1=DICT#(QNAME);
   IF I1=0 THEN
          DO;
   /* PRINT INCOMPLETENESS ERROR*/
          CALL PINCERR(QNAME,CURRENT_ASSERT_NAME);
          RETURN;



          END;
   #LABELS=#LABELS + 1;
   IF #LABELS> MAX#_LABELS THEN CALL SYSERR
   ('ENEXDP:MAX# LABELS EXCEEDED');
   LABEL(#LABELS)= I1;
0         END;  /* END OF "REPLACE"  CASE  */
-      ELSE  /* CHECK IF  ASSERTION USES A CONDITIONAL FUNCTION,
                IN THE CASE THAT QNAME IS TARGET TO ASSERTION,
                AND IF SO SET THE DEPENDENCY CODE TO SHOW A CONDITIONAL
                DEPENDENCY  */
          IF ST_IND=1 & COND(FCN)     THEN                                  4
             CALL ENDRMAT(QNAME,CURRENT_ASSERT_NAME,COND_DEP_CODE);          5
-         ELSE /*NORMAL ASSERTION: ENTER EXPLICIT DEPENDENCY RELATIONSHIP
          IN MATRIX IF SOURCE/TARGET NAME RETURNED VALIDATED */
C            CALL ENDRMAT(QNAME,CURRENT_ASSERT_NAME,EXPL_DEP_CODE);
0  /* DEPENDENCY BETWEEN ASSERTION AND QNAME HAS BEEN ENTERED;
           EITHER EXPLICIT DEP, COND DEPENDENCE, OR AUTO. DEP  */
```

```
-        /* IN ADDITION, CHECK IF 'QNAME' IS THE TARGET OF THE
          ASSERTION (ST_IND=1) & QNAME IS A 'SUBSET.X' TYPE TARGET
          AND IF X IS A TARGET FILE ==>> THAT THIS
          IS A SUBSET DESCRIPTION), THEN ENTER THE AUTO. DEPEPNCENCY
          CODE IN THE MATRIX BETWEEN THE 'SUBSET.X' NAME AND 'R',
          WHERE 'R' IS THE RECORD THAT CORRESPONDS TO 'X', THE
          TARGET FILE NAME. DONT NEED TO WORRY ABOUT X IF IT IS A
          SOURCE FILE, BECAUSE BY THE PRECEDENCE SCHEME IT WILL
          FOLLOW WHATEVER IT DEPENDS ON  */
C        IF ST_IND=1 THEN
         CHECK_SUBSET_CASE:                                           14- 18
         DO;
            IF LENGTH(QNAME) > 7 THEN
            DO;
               IF SUBSTR(QNAME,1,7)            ='SUBSET.' THEN
              DO;
                 TEMP_TAR_FILE=SUBSTR(QNAME,8);
                 J=DICT#(TEMP_TAR_FILE);
                 IF J=0 THEN CALL PNETERR(
                  '(INCONSISTENCY): SUBSET OF FILE "'||TEMP_TAR_FILE||
                  '" DESCRIBED, BUT NO CORRESPONDING FILE DESCRIBED');
                 ELSE /* WE CHECK IF SUBSET FILE IS A TARGET FILE & IF
                            SO, ENTER CODE IN MATRIX BETWEEN SUBSET NAME
                         AND THE CORRESPONDING RECORD NAME; BUT MUST
                         FIRST FIND RECNAME THAT CORRESPONDS TO FILE */
                 DO;
                   IF LENGTH(TEMP_TAR_FILE)>4 THEN
                    IF SUBSTR(TEMP_TAR_FILE,1,4)='OLD.'|
                        SUBSTR(TEMP_TAR_FILE,1,4)='NEW.' THEN
                    BARE_FILE_NAME=SUBSTR(TEMP_TAR_FILE,5);
                    ELSE BARE_FILE_NAME=TEMP_TAR_FILE;
                   ELSE BARE_FILE_NAME=TEMP_TAR_FILE;
                   FILE_ST_TYPE=FILEST(BARE_FILE_NAME);
                   IF FILE_ST_TYPE='TG' |
                      (FILE_ST_TYPE='ST' & SUBSTR(TEMP_TAR_FILE,1,4)=
                        'NEW.')  /* I.E. SUBSET FOR A TARGET FILE  */
                   THEN
                   DO;
                      FILENAME_RETREVE='$P'||BARE_FILE_NAME;
                   CALL RETREVE($RECD||'&'||FILENAME_RETREVE||'|'||
                     '$PTN||'&'||FILENAME_RETREVE,'',
                     START,REC_PTR,#REC_PTR);
                   IF #REC_PTR ¬=1 THEN CALL SYSERR(
                          'ENEXCP: SUBSET REC NOT FND');
                   P=REC_PTR(1);
                   RECNAME=CHPTELP(P->NAME(1));



                   IF SUBSTR(TEMP_TAR_FILE,1,4)='NEW.' THEN
                      RECNAME='NEW.'||RECNAME;
                   K=DICT#(QNAME);                                    16
                   L=DICT#(RECNAME);                                  17
                   ADJMAT(K,L)=AUTO_DEP_CODE;                         18
           END CHECK_SUBSET_CASE;
-        END;  /* END OF TRYING TO ENTER DEPENDENCY RELATIONSHIP
                   BETWEEN THIS ASSERTION AND THIS QNAME  */
0   /* NOW LOOK AT NEXT NAME OF THIS ASSERTION  */
-         NAMEIND=NAMEIND+NCOMPONENTS(J);                             19
-      END;     /* END OF PROCESSING DEPENDENCY OF ALL NAMES TO THIS  20
                ASSERTION  */
```

```
-END GTASNMS;
1ENDRMAT: PROC(ONAME,CURASNM,DEPCODE);
   /* THIS PROCEDURE ENTERS THE DEPENDENCY RELATIONSHIP WITH        5-8
      CODE "DEPCODE" BETWEEN THE
      ITEM 'ONAME' AND THE ASSERTION 'CURASNM' IN THE 'ADJMAT';
      IF ONAME IS SOURCE OF ASSERTION THEN RELATIONSHIP IS FROM
      ONAME TO CURASNM; IF ONAME IS TARGET THEN VICE VERSA   */
     DCL ONAME CHAR(MAX_LEN_ONM)  VAR;
     DCL CURASNM CHAR(MAX_LEN_NAME)VAR;
 DCL DEPCODE FIXED DEC(1);                                          5
     K=DICT#(ONAME);
     IF K=0 THEN   CALL PINCERR(ONAME,CURASNM);                     6
     ELSE DO;
     L= DICT#(CURASNM);                                             7
     IF L=0 THEN CALL SYSERR ('ENEXCP: '||CURASNM||' NOT IN DICT');
     IF ST_IND=0 THEN /* ONAME IS A SOURCE TO THE ASSERTION */
     ADJMAT(K,L)=DEPCODE;                                           8
 ELSE ADJMAT(L,K)= DEPCODE;
     END;
0END ENDRMAT;
1PINCERR: PROC(ONAME,CURASNM);                                      9
 DCL ONAME CHAR(*) VAR,  CURASNM CHAR(*);
  DCL MSG_PREFIX CHAR(38)  INIT(
                '(INCOMPLETENESS): NEED TO KNOW HOW TO ' )  STATIC;
   DCL MSG_SUFFIX  CHAR(65) VAR;
        /* ONAME IS NOT IN DICTIONARY; I.E. UNDEFINED */
     MSG_SUFFIX=' "'||ONAME||'" IN ASSERTION "'||CURASNM||'"';
        IF ST_IND=0 THEN /* ONAME IS AN UNDEFINED SOURCE TO THE
           ASSERTION  */
        CALL PNETERR(MSG_PREFIX||'OBTAIN'||MSG_SUFFIX);
        ELSE  /* ONAME IS AN UNDEFINED TARGET OF ASSERTION */
        CALL PNETERR(MSG_PREFIX||'USE'||MSG_SUFFIX);
 END PINCERR;
-END ENEXCP;
```

```
*PROCESS ('NST,MACRO,N=ENHRREL,SM=(2,72,1)');
 ENHRREL: PROC;      /* ENTER ALL THE HIERARCHICAL REALATIONSHIPS INTO
      THE ADJACENCY MATRIX 'ADJMAT' */
      %DCL MAX_LEN_QNAME FIXED;
     %DCL MAX#_TOT_FILES FIXED;
      %DCL MAX_LEN_NAME FIXED;
      %DCL MAX#_RETPTRS FIXED;
      %DCL HIER_INPUT_CODE FIXED;
      %DCL HIER_OUTPUT_CODE FIXED;
      %MAX_LEN_QNAME=32;
      %MAX#_TOT_FILES=12;
      %MAX_LEN_NAME=10;
      %MAX#_RETPTRS=200;
      %HIER_INPUT_CODE=1;
      %HIER_OUTPUT_CODE=2;
      %INCLUDE INCLIB (CSECIR);


      DCL ADJMAT(DICTIND,DICTIND)FIXED BIN CTL EXT;
      /*ALREADY ALLOCATED BY NETGEN */
      DCL DICTIND FIXED BIN EXT;
      DCL START POINTER EXT;
      DCL (PQNAME,DQNAME)CHAR(MAX_LEN_QNAME)VAR;
      DCL RETPTRS(MAX#_RETPTRS)POINTER EXT;
      DCL (FILENAME,RECNAME,STORNAME,KEYNAME)CHAR(MAX_LEN_NAME);
      DCL CHPTHLP ENTRY(CHAR(*))RETURNS(CHAR(MAX_LEN_QNAME)VAR);
      DCL FILE_S_T_TYPE CHAR(2) STATIC;
      DCL FILETYPE CHAR(MAX_LEN_NAME)STATIC INIT('$FILE');
      DCL RPTNTYPE CHAR(MAX_LEN_NAME)STATIC INIT('$RPTN');
      DCL RECDTYPE CHAR (MAX_LEN_NAME) STATIC INIT ('$RECD');
      DCL REPTTYPE CHAR(MAX_LEN_NAME) STATIC INIT('$REPT');
      DCL GRPTYPE  CHAR(MAX_LEN_NAME)STATIC INIT('$GRP ');
      DCL FLDTYPE  CHAR(MAX_LEN_NAME)STATIC INIT('$FLD ');
      DCL ASSERT_TYPE CHAR(MAX_LEN_NAME) STATIC INIT('$ASSERT');
      DCL NOT CHAR(1) INIT('¬') STATIC;
      DCL DOT CHAR(1) STATIC  INIT('.');
      DCL AND CHAR(1) INIT('&') STATIC;
      DCL OR CHAR(1) INIT ('|') STATIC;
      DCL FILEST ENTRY (CHAR(MAX_LEN_NAME))RETURNS(CHAR(2));
      DCL DICT# ENTRY(CHAR(*)VAR)  RETURNS(FIXED BIN);
      DCL PARENT ENTRY(POINTER) RETURNS(CHAR(MAX_LEN_NAME));
      DCL #FILEDEFS FIXED BIN;
      DCL FILE_DEF_PTR(MAX#_TOT_FILES) POINTER;
      /* GET EACH FILE OR REPORT DEFINITION STORAGE ENTRY  */
      CALL RETREVE(FILETYPE||OR||REPTTYPE,'',START,FILE_DEF_PTR,                 1
                #FILEDEFS);
      DO I=1 TO #FILEDEFS;                                                        2-3
        /* FOR EACH FILE OR REPORT DEFINITION */
           STORAGE_PTR=FILE_DEF_PTR(I);
           FILENAME=NAME(1);
           RECNAME=NAME(2);
           STORNAME=NAME(3);
           KEYNAME=NAME(4);
           FILE_S_T_TYPE=FILEST(FILENAME);  /*   CURRENT FILE IS SET TO
             'SE' (SOURCE), 'TG' (TARGET)  OR 'ST' (BOTH)  */
           CALL ENT_HIER_ADJ(FILENAME,RECNAME);                                   4
           CALL ENT_MED_ADJ(FILENAME,STORNAME);                                   5
      END;
```

```
IENT_HIER_ADJ: PROC(PAR,DESCENDANT)  RECURSIVE;
      /*THIS PROC ENTERS THE HIERARCHICAL RELATIONSHIP BETWEEN 'PARENT'
      NAME AND 'DESCENDANT' NAME IN THE ADJACENCY MATRIX*/
      /* THE TYPE OF RELATIONSHIP IS AS FOLLOWS:
      1= HIERARCHICAL RELATIONSHIP IF THE TWO  ITEMS ARE IN A SOURCE
      FILE, AND RELATION GOES FROM PARENT TO DESCENDANT;
      2=HIERARCHICAL RELATIONSHIP IF THE TWO ITEMS ARE IN A TARGET FILE
              AND RELATION GOES FROM DESCENDANT TO PARENT.
      IF THE CURRENT PARENT FILE IS BOTH A SOURCE AND TARGET FILE, THEN
      THEN THE TWO ITEMS ARE PREFIXED WITH 'OLD' AND 'NEW' AND THE
      RELATION IS ENTERED IN BOTH DIRECTIONS, WITH TYPE 1 AND 2
      RESPECTIVELY*/
      %DCL MAX#_APPEAR_FLD FIXED;
      %MAX#_APPEAR_FLD=50;
      DCL RET_PAR_NAME CHAR(MAX_LEN_NAME);
      DCL RET_DESC_STR CHAR(33) VAR;
      DCL PAR CHAR(*);
      DCL DESCENDANT CHAR(*);
      DCL (PDNAME,DDNAME) CHAR(MAX_LEN_DNAME)VAR;
      DCL CURRENT_SE POINTER;
      /*THE CURRENT STORAGE ENTRY IS POINTED TO BY 'CURRENT_SE'*/
      DCL DESC_PTR(MAX#_APPEAR_FLD) POINTER;
      DCL #DESC_PTR FIXED BIN;



      DCL DF POINTER;
      DCL (I,J,LAST_DESC_POS)  FIXED BIN;
      % INCLUDE INCLIB(DANY);
      /*FURTHERMORE, THIS PROCEDURE ENTERS THE RELATIONSHIPS BETWEEN
      EACH DESCENDANT AND ITS SUCCESSIVE DESCENDANTS BY CALLING ON ITSELF
      RECURSIVELY*/
      CALL SYNFMAT;                                                            1 - 7
      RET_PAR_NAME='$P'||FILENAME;    /* SET PARENT NAME */
        /* RETRIEVE STORAGE ENTRY WHERE DESCENDANT ITSELF IS DEFINED */
      RET_DESC_STR=DESCENDANT||AND||RET_PAR_NAME;                              8
        CALL RETREVE(RET_DESC_STR
             ,'',START,DESC_PTR,#DESC_PTR);
       /* EXPECT 2 STORAGE ENTRIES WITH DESCENDANT NAME: ONE WHERE
         MENTIONED IN PARENT DEFINITION AND ONE WHERE DESCENDANT
         ITSELF IS DEFINED  */
      IF #DESC_PTR =0  THEN                                                    9
      DO;
          CALL PRHERR(08);                                                     14
          RETURN;
      END;
      CURRENT_SE,
      STORAGE_PTR=DESC_PTR(1);
      IF NAME(1)¬=DESCENDANT THEN
      DO;
         IF #DESC_PTR < 2 THEN
         DO;
          CALL PRHRERR(0B);                                              14
          RETURN;
         END;
         IF #DESC_PTR >2  THEN
         DO;
          CALL PRHRERR(1B);                                              15
          RETURN;
         END;
         CURRENT_SE,
         STORAGE_PTR=DESC_PTR(2);
         IF NAME(1)¬=DESCENDANT THEN CALL SYSERR(DESCENDANT||' UNDEF');
      END;
         DP=DATA_PT;
         IF ANY_STMT.TYPE='FLD '  THEN     ;                              10-13
         ELSE
        DO;
          IF      ANY_STMT.TYPE='GRP ' | ANY_STMT.TYPE='RECD' |           10
                    ANY_STMT.TYPE='RPTN' THEN
          DO;
            LAST_DESC_POS=#KEYS - 2; /* LAST DESCENDANT NAME IS IN
               POSITION #KEYS-2 OF THE CURRENT STORAGE ENTRY  */
            DO I=2 TO LAST_DESC_POS;                                      11
              STORAGE_PTR=CURRENT_SE;
              CALL ENT_HIER_ADJ(NAME(1),NAME(I));                         12
            END;
          END;
          ELSE CALL SYSERR(DESCENDANT||' ILLEGAL TYPE');
        END;
 1PRHRERR:  PROC(UNDEF_OR_AMBIG);
      DCL UNDEF_OR_AMBIG  FIXED BIN;
      DCL MSG(0:1) CHAR(25) VAR INIT(' MISSING',
           ' DESCRIBED MORE THAN ONCE');
      DCL MSG_TYPE(0:1) CHAR(14) VAR INIT('INCOMPLETENESS',
                'I'CONSISTENCY');
      CALL PRETERR('('||MSG_TYPE(UNDEF_OR_AMBIG)||'): GROUP OR FIELD '||
           DESCENDANT||' IN '||PAR||MSG(UNDEF_OR_AMBIG));
      RETURN;


  END PRHPERR;
 1ENHRMAT: PROC;
      /*QUALIFY PARENT AND DESCENDANT*/                                    1
      IF PAR=FILENAME THEN
      DO;
         PONAME=CHPTRLB(FILENAME);
         DONAME=CHPTRLB(RECNAME);
      END;
      ELSE
      DO;
         IF PAR=RECNAME THEN PONAME=CHPTRLB(PAR);   /* RECNAMES AND RPTN NAMES
                       REMAIN UNQUALIFIED SINCE THEYRE UNIQUE  */
         ELSE     /* QUALIFY THE PARENT GROUP NAME */
         PONAME=CHPTRLB(PARENT(STORAGE_PTR))||DCT||CHPTRLB(PAR);
         /* QUALIFY THE DESCENDANT NAME:    */
         DONAME=CHPTRLB(PARENT(STORAGE_PTR))||DCT||CHPTRLB(DESCENDANT);
      END;
      IF FILE_S_T_TYPE='SR'|FILE_S_T_TYPE='TG' THEN
      DO;
         I=DICT#(PCNAME);                                                   2
         IF I=0 THEN CALL  SYSERR(PCNAME||' NOT IN DICT');
         J=DICT#(DCNAME);                                                   3
         IF J=0 THEN CALL  SYSERR(DCNAME||' NOT IN DICT');
      END;
      IF FILE_S_T_TYPE='SR'  THEN                                           4
      /*CURRENT FILENAME IS A SOURCE FILE*/
         ADJMAT(I,J)=HIER_INPUT_CODE;                                       5
      ELSE IF FILE_S_T_TYPE='TG' THEN
         ADJMAT(J,I)=HIER_OUTPUT_CODE;                                      6
      ELSE  IF FILE_S_T_TYPE='ST' THEN                                      4
      DO;  /*CURRENT FILE NAME IS BOTH INPUT AND OUTPUT FILE*/
         I=DICT#('OLD.'||PCNAME);                                           7
         IF I=0 THEN CALL SYSERR('OLD.'||PCNAME||' NOT IN DICT');
         J=DICT#('OLD.'||DCNAME);
         IF J=0 THEN CALL  SYSERR('OLD.'||DCNAME||' NOT IN DICT');
         ADJMAT(I,J)=HIER_INPUT_CODE;
         I=DICT#('NEW.'||PCNAME);
         IF I=0 THEN CALL SYSERR('NEW.'||PCNAME||' NOT IN DICT');
         J=DICT#('NEW.'||DCNAME);
         IF J=0 THEN CALL SYSERR('NEW.'||DCNAME||' NOT IN DICT');
         ADJMAT(J,I)=HIER_OUTPUT_CODE;
       END;
      ELSE CALL SYSERR(FILENAME||' NOT S/T');
  0END ENHRMAT;
  -END ENT_HIER_ADJ;
  1ENT_MED_ADJ: PROC(FLNAME,STNAME);
      /*THIS PROC ENTERS THE RELATIONSHIP BETWEEN A FILENAME (FLNAME)
      AND ITS CORRESPONDING STORAGE MEDIUM NAME (STNAME) AND ENTERS IT
      IN THE ADJACENCY MATRIX AS RELATIONSHIP TYPE 6 */
      #DCL FILE_STOR_CODE FIXED;
      #FILE_STOR_CODE=6;
      DCL (FLNAME,STNAME) CHAR(*);
      DCL (TEMP_FLNAME,OLD_TEMP_FLNAME,NEW_TEMP_FLNAME)  CHAR
            (MAX_LEN_CNAME)  VAR;
      DCL (I,J) FIXED BIN;
      TEMP_FLNAME=CHRTRLB(FLNAME);
      IF STNAME=' ' THEN RETURN;
      J=DICT#(CHRTRLB(STNAME));
      IF J=0 THEN CALL  SYSERR('STORAGE NM '||STNAME||' NOT IN DICT');
      IF FILE_S_T_TYPE ='ST' THEN  /* FILE IS BOTH SOURCE & TARGET */
      DO;
         OLD_TEMP_FLNAME='OLD.'||TEMP_FLNAME;
         I=DICT#(OLD_TEMP_FLNAME);



         IF I=0 THEN CALL  SYSERR('FILENM '||OLD_TEMP_FLNAME||
                ' NOT IN DICT');
         ADJMAT(I,J)=FILE_STOR_CODE;
         NEW_TEMP_FLNAME='NEW.'||TEMP_FLNAME;
         I=DICT#(NEW_TEMP_FLNAME);
         IF I=0 THEN CALL  SYSERR('FLNAME '||NEW_TEMP_FLNAME    ||
                      ' NOT IN DICT');
         ADJMAT(I,J)=FILE_STOR_CODE;
      END;
      ELSE  /* FILE IS EITHER SOURCE OR TARGET  */
      DO;
         I=DICT#(TEMP_FLNAME);
         IF I=0 THEN CALL  SYSERR('FLNAME '||TEMP_FLNAME||' NOT IN DICT');
         ADJMAT(I,J)=FILE_STOR_CODE;
      END;
  0END ENT_MED_ADJ;
  -END ENHREL;
```

```
*PROCESS ('NST,MACRO,N=ENPTREL');
 ENPTREL:PROC;
     /*THIS PROCEDURE ENTERS EACH POINTER NAME TO POINT TO ITS
     CORRESPONDING RECD IN THE ADJACENCY MATRIX*/
     %DCL MAX_LEN_NAME FIXED;
     %DCL MAX_LEN_QUAL_NAME  FIXED;
     %DCL PTR_REL_CODE FIXED;
     %DCL MAX#_POINTER_NAMES FIXED;
     %MAX_LEN_NAME=10;
     %MAX#_POINTER_NAMES=12;
     %MAX_LEN_QUAL_NAME =32;
     %PTR_REL_CODE=5;
     % INCLUDE INCLIB(DDICT);
     DCL ADJMAT(DICTIND,DICTIND) FIXED BIN EXT CTL;
     DCL DICT# ENTRY (CHAR(*)VAR)RETURNS(FIXED BIN);
     DCL RCNAME CHAR(MAX_LEN_QUAL_NAME) VAR;
     DCL QRECNAME CHAR(MAX_LEN_QUAL_NAME) VAR;
     DCL START POINTER EXT;
     /* FIND ALL POINTER NAMES IN THE DICTIONARY:  I.E. NAMES BEGINNING WITH
                 'POINTER.'    */
     DO I=1 TO DICTIND WHILE (DICT(I)<'FF');                                  1
        IF LENGTH(DICT(I))>8   THEN
         IF SUBSTR(DICT(I),1,9)='POINTER.' THEN                               2
         DO;
            /* TAKE CURRENT DICTIONARY ENTRY BEGINNING WITH 'POINTER.'
                AND EXTRACT THE RECORD NAME WHICH FOLLOWS: THE RECORD NAME
                CAN BE QUALIFIED WITH A PREFIXED 'OLD.' OR 'NEW.'   */         3
            QRECNAME=SUBSTR(DICT(I),9);
                J=DICT#(QRECNAME);                                            4
                IF J=0 THEN CALL SYSERR(QRECNAME||' NOT GOOD PTRNM');
                ADJMAT(I,J)=PTR_REL_CODE;                                     5
         END;
     END;
OEND ENPTREL;
```

```
*PROCESS('M,A,X,SM=(2,72,1),N=FLOWCPT,EXTREF');
 FLOWCPT: PROC;
 /* THIS PROCEDURE TAKES THE ORDER VECTOR AND FINDS THE LOCATION OF
 'DO END' PAIRS. */
 /* REPEATING GROUPS IN THE 'MODEL' LANGUAGE WILL REQUIRE THAT
 'DO' LOOPS BE GENERATED. THESE MAY BE NESTED AND THEIR RANGES
 MUST BE DISCOVERED. TO MAKE SURE THAT THIS NESTING TAKES PLACE
 PROPERLY AND THAT STATEMENTS NOT DEPENDING ON THE LOOP ARE NOT IN ITS
 RANGE THE ORDER VECTOR WILL BE REARRANGED. CARE MUST BE EXERCISED
 SO AS NOT TO UPSET ANY PRECEDENCE RELATIONS. */
0/* EXTERNAL PROCEDURES. */
 /* WHERE DETERMINES HOW FAR A NODE IN THE ORDER VECTOR HAS BEEN
 DISPLACED WHEN NODES HAVE BEEN MOVED OUTSIDE RANGES OF DO LOOPS. */
 DCL WHERE ENTRY (FIXED BIN, (*) FIXED BIN, (*) FIXED BIN, FIXED BIN,
                  FIXED BIN) RETURNS(FIXED BIN);
 /* RECTAR TAKES A NODE NUMBER AND A 'FOREACH' NAME AND RETURNS A
 POINTER TO A LINKED LIST OF STRUCTURES LIKE 'MOVE_REC_FLD'. THIS
 LIST CONTAINS THE NAMES OF FILES AND RECORDS WHICH CONTAIN FIELDS
 NAMED AS TARGETS OF THE ASSERTION. A FILE OR RECORD IS INCLUDED
 ONLY IF THE FIELD IT CONTAINS IS MODIFIED BY THE GIVEN 'FOREACH' NAME.
                                                                    */
 DCL RECTAR ENTRY RETURNS (PTR);
 7DCL EACH_NAME_LEN FIXED;
 3EACH_NAME_LEN=13;
0/* INTERNAL PROCEDURES----- REMOVE THESE DECLARATIONS FOR OPT COM. */
 DCL FOREACH ENTRY(FIXED BIN, FIXED BIN) RETURNS(PTR);                ********
 DCL BADOVER ENTRY((*) FIXED BIN, FIXED BIN) RETURNS(BIT(1));         *******



 DCL NEW_POS ENTRY RETURNS(FIXED BIN(15));                            $$$$$$$
 DCL MOVE_OUT ENTRY RETURNS(BIT(1));                                   $$$$$$$
 DCL AMONG ENTRY RETURNS (BIT(1));                                     $$$$$$
0/* EXTERNAL VARIABLES. */
 /* PATHMAT IS THE PATH MATRIX FOR ALL NODES. */
 DCL DEP_NODE FIXED BIN;
 DCL PATHMAT (*,*) BIT(1) CTL EXT;
 %INCLUDE INCLIB(DDICT);
 DCL ORDER(*) FIXED BIN EXT CTL;
 /* 'DOTAB' AND 'ENDTAB' ARE ARRAYS USED TO COMMUNICATE THE LOCATION
 OF THE 'DO' AND 'END' STATEMENTS. THEY HOLD THE SUBSCRIPTS OF
 OF THE FIRST AND LAST ELEMENTS OF THE ORDER VECTOR IN THE LOOP.
 ADDITIONALLY 'DOTAB' CONTAINS THE 'FOREACH' NAME USED IN THE LOOP.
 BEFORE RETURN THESE LISTS ARE SORTED  INTO ASCENDING ORDER
 THIS INSURES THAT AS THE ORDER VECTOR IS SCANNED THESE LISTS CAN
 BE MERGED INTO IT RATHER THAN CHECKED COMPLETELY AT EACH NODE. */
```

```
-/* WE CHECK EVERY ELEMENT IN THE ORDER VECTOR. */
 #LOOPS=0;                                                                1
 START_PTR=NULL;
 CHK_ORDER: DO VEC_ENTRY#=1 TO DICTINC;
      NODE#=ORDER(VEC_ENTRY#);
      /* ONLY ASSERTIONS ARE TO BE CHECKED. */
0     YES_ASSN: IF DICTYPE(NODE#)='ASTC' THEN DO;                         2
           EACH_PTR =FOREACH(NODE#,VEC_ENTRY#);                           3
           IF EACH_PTR¬= NULL THEN DO;                                    4
                /* WE MUST GENERATE A DO END FOR THIS FOREACH NAME. */
                /* FIND THE NODE FARTHEST DOWN IN THE ORDER
                 VECTOR WHICH IS DEPENDENT ON THIS NODE. THIS WILL BE
                THE LAST NODE IN THE LOOP. */
                DO LAST_DEP=DICTINC TO 1 BY -1 WHILE(¬PATHMAT(NODE#,      5
                                                  ORDER(LAST_DEP)));
                END;
0               /* PARTITION THE PORTION OF THE ORDER VECTOR BETWEEN
                THIS NODE AND ITS FURTHEST DESCENDENT INTO TWO GROUPS:
                THOSE IN THE LOOP AND THOSE OUTSIDE. */
                #NOT=0;
                #YES=1;
                YES_DEP(1)=VEC_ENTRY#;
                SAVE_ORDER_DEP(1)=NODE#;
                PARTITION:DO PART_NODE#=VEC_ENTRY#+1 TO LAST_DEP;         6
                     /* IS THIS NODE DEPENDENT ON THE ASSERTION? IF SO
                     IT IS IN THE LOOP. SAVE ITS LOCATION AND VALUE. */
                      DEP_NODE=ORDER(PART_NODE#);
                      IF PATHMAT(NODE#,DEP_NODE   ) THEN DO;
                           #YES=#YES+1;
                           YES_DEP(#YES)=PART_NODE#;
                           SAVE_ORDER_DEP(#YES)=DEP_NODE;
                      END;
                      ELSE DO;
                           #NOT=#NOT+1;
                           NOT_DEP(#NOT)=PART_NODE#;
                      END;
                END PARTITION;
0               /* TEST THE NEW LOOP FOR ILLEGAL OVERLAPS WITH OLD
                LOOPS. */
                IF ¬BADOVER(YES_DEP,#YES) THEN DO;                        7
0                    /* ALL O. K. MOVE THE NODES. */
                     /* THE ONES NOT DEPENDENT MUST GO FIRST. */
                     DO MOVE#=1 TO #NOT;                                  8
                          ORDER(MOVE#+VEC_ENTRY#-1)=ORDER(NOT_DEP(
                                    MOVE#));
                          END;
                     DO MOVE#=1 TO #YES;
                          ORDER(MOVE#+VEC_ENTRY#+#NOT-1)=SAVE_ORDER_DEP(
                                    MOVE#);
                     END;
0                    /* WE HAVE MOVED NODES SO WE MUST UPDATE ANY ALTERED
                     DO END PAIRS. */
                     /* CHECK THE LINKED LIST. */
                     T_LIST_PTR=START_PTR;
                     DO WHILE (T_LIST_PTR¬=NULL);
                          DO_END_PTR=T_LIST_PTR;
                          DISP=WHERE(DO_LOC,NOT_DEP, YES_DEP, #NOT,#YES);
                          DO_LOC=DO_LOC+DISP;
                          END_LOC=END_LOC+DISP;
                          T_LIST_PTR=NEXT_DO_END;
                     END;
0                    /* SAVE THE LOCATIONS OF THE DO AND END. */
                     YES_DEP(1)=VEC_ENTRY#+#NOT;
```

```
                          YES_DEP(#YES)=VEC_ENTRY#+#ACT+#YES-1;
                     END;
                     /* HERE WE HAVE AN ILLEGAL OVERLAP. */
                     ELSE;  /* THE ERROR MESSAGE IS DONE BY BADOVER. */
0                    /* NO MATTER WHAT ADD ALL LOOPS STARTING AT THIS POINT
                     NOTE THAT THEY ALL HAVE THE SAME BOUNDS. */
                     DO WHILE (EACH_PTR¬=NULL);
                          NAME_PTR=EACH_PTR;
                          ALLOCATE DO_END_LIST;
                          NEXT_DO_END=START_PTR;
                           DO_LOC=YES_DEP(1);
                           END_LOC=YES_DEP(#YES);
                           LOOP_VAR=FORE_NAME;
                           START_PTR=DO_END_PTR;
                           #LOOPS=#LOOPS+1;
                           EACH_PTR=NEXT_NAME;
                           FREE NAME_LIST;
                     END;
0                    /* ALSO DECREMENT 'VEC_ENTRY# SO  THE MOVED NODES WILL
                     BE CHECKED FOR LOOPS. */
                     VEC_ENTRY#=VEC_ENTRY#-1;
                END;
           END;
 END CHK_ORDER;
0IF #LOOPS=0 THEN RETURN;
-/* IF AN OUTPUT RECORD OR FILE CONTAINS A FIELD WHICH IS
 MODIFIED BY A LOOP VARIABLE THE RECORD OR FILE SHOULD NOT BE
 OUTPUT UNTIL AFTER THE LOOP. THIS ALLOWS THE LOOP TO COMPLETE THE
 FIELD AND PREVENTS MULTIPLE_OUTPUT. */
 /* EACH LOOP MUST BE CHECKED. */
 DO_END_PTR=START_PTR;
0DO WHILE(DO_END_PTR¬=NULL);                                              11
      TOP_PTR=NULL;
      #MOVED=0;
      #MOVED_ORDER=0;
      /* EACH NODE IN THE RANGE OF THE LOOP MUST VE CHECKED. */
0     DO VEC_ENTRY#=DO_LOC TO END_LOC;                                    12
           NODE#=ORDER(VEC_ENTRY#);
           NODE_TYPE=DICTYPE(NODE#);
           /* WE MUST KEEP TRACK OF RECORDS AND FILES CONTAINING
           TARGET FIELDS WHICH ARE MODIFIED BY 'FOREACH'. */
0          IF NODE_TYPE='ASTG' THEN DO;
                /* FIND THE RECORDS AND FILES CONTAINING THE TARGETS. */
                REC_PTR=RECTAR(NODE#,LOOP_VAR);
                /* CHAIN THE NEW ONES TO THOSE PREVIOUSLY FOUND
                   BY HOOKING THE NEW LIST ON THE FRONT OF THE OLD. */
                IF REC_PTR¬=NULL THEN DO;                                   13
                      MOVE_PTR=REC_PTR;
                      DO WHILE(NEXT_PAIR¬=NULL);
                           MOVE_PTR=NEXT_PAIR;
                      END;
                      NEXT_PAIR=TOP_PTR;
                      TOP_PTR=REC_PTR;
                END;
           END;
           ELSE DO;
0               /* IF THE NODE IS A 'RECD' OR 'FILE' WE MUST CHECK
                TO SEE IF IT CONTAINS A FIELD WHICH IS MODIFIED
                BY A 'FOREACH' NAME AND IS THE TARGET OF A PREVIOUS
                ASSERTION. TO DO THIS 'MOVE_OUT' CHECKS THE LIST,
                'MOVE_REC_FLD', WE HAVE BEEN ACCUMULATING. NOTE THAT
                THE PRECEDENCE RELATION INSURES THAT ANY FIELDS IN A
                RECORD OR FILE MUST BE FILLED BEFORE IT IS OUTPUT.
```

```
            THUS WE DO NOT NEED TO PASS THROUGH THE ORDER VECTOR
            TWICE. */
            IF ((NODE_TYPE='RECD')|(NODE_TYPE='RPTN')) THEN DO;        14
                IF MOVE_OUT THEN DO;
                /* SAVE THE LOCATION IN 'ORDER' OF THE NODES TO
                BE MOVED. */
                    #MOVED=#MOVED+1;
                    MOVE_NODE(#MOVED)=VEC_ENTRY#;
                END;
            END;
            ELSE
            IF    NODE_TYPE='FILE' |NODE_TYPE='REPT'    THEN DO;       14
                IF MOVE_OUT THEN DO;
                   #MOVED=#MOVED+1;
                    MOVE_NODE(#MOVED)=VEC_ENTRY#;
                END;
            END;
        END;
    END;  /* END OF ITERATION THROUGH LOOP. */
0   /* NOW CHECK TO SEE IF NODES ARE MOVED. */
    IF #MOVED>0 THEN DO;                                            16
        /* MOVE THE NODES. */
            /* 'NEXT_LOC' INDICATES THE POSITION AVAILABLE FOR THE
            NEXT NODE IN THE LOOP. */
        NEXT_LOC=DO_LOC;
        DO VEC_ENTRY#=DO_LOC TO END_LOC;
            /* 'AMONG' CHECKS TO SEE IF 'VEC_ENTRY#' IS AMONG
            THE NODES TO BE MOVED. */
            IF AMONG THEN DO;
                /* SAVE THE NODES TO BE MOVED. */
                #MOVED_ORDER=#MOVED_ORDER+1;
                MOVED_ORDER(#MOVED_ORDER)=ORDER(VEC_ENTRY#);
            END;
            ELSE DO;
                /* MOVE NODE UP. */
                ORDER(NEXT_LOC)=ORDER(VEC_ENTRY#);
                NEXT_LOC=NEXT_LOC+1;
            END;
        END;
0       /* INSERT THE MOVED NODES. */
        DO J=1 TO #MOVED;
            ORDER(NEXT_LOC+J-1)=MOVED_ORDER(J);
        END;
0           /* CHANGE THE END LOCATION THE START IS O. K. */
            END_LOC=END_LOC-#MOVED;
        /* WE HAVE MOVED NODES - UPDATE DO END LOCATIONS. */
        SAVE_PTR=DO_END_PTR;
        DO_END_PTR=START_PTR;
        DO WHILE(DO_END_PTR¬=NULL);
                /* WE MUST NOT UPDATE THE CURRENT LOOP TWICE. */
                IF DO_END_PTR¬=SAVE_PTR THEN DO;
            /* 'NEW_POS' COMPUTES ANY DISPLACEMENT CAUSED BY THE
            SHIFT OF NODES. */
            DO_LOC=DO_LOC+NEW_POS(DO_LOC);
            END_LOC=END_LOC+NEW_POS(END_LOC);
                END;
            DO_END_PTR=NEXT_DO_END;
        END;
        DO_END_PTR=SAVE_PTR;
    END;
0   /* FREE THE STORAGE ASSOCIATED WITH THE LIST OF TARGETS. */
    MOVE_PTR=TOP_PTR;
    DO WHILE(MOVE_PTR¬=NULL);
```

```
          SAVE_PTR=NEXT_PAIR;
          FREE MOVE_REC_FLD;
          MOVE_PTR=SAVE_PTR;
     END;
     DO_END_PTR=NEXT_DO_END;
 END;
-/* WE MUST UPDATE THE RANK VECTOR TO REFLECT THE REORDERING THAT
 HAS BEEN DONE. */
 /* WE ARE NOT ELEGANT. SIMPLY REDO THE RANK VECTOR SO THAT ADJACENT
 ELEMENTS IN THE NEW ORDER VECTOR WHICH HAVE THE SAME OLD RANKS HAVE
 EQUAL NEW RANKS. WE LOSE TRACK OF SOME PARALLELISM HOWEVER. */
 CUR_ORDER=ORDER(1);
 PREV_RANK,CUR_RANK=RANK(CUR_ORDER);
 NEW_RANK(CUR_ORDER)=CUR_RANK;
 DO J=2 TO DICTIND;                                                  18 ~ 21
     CUR_ORDER=ORDER(J);
     NXT_RANK=RANK(CUR_ORDER);
     IF NXT_RANK¬=PREV_RANK THEN CUR_RANK=CUR_RANK+1;               19
     NEW_RANK(CUR_ORDER)=CUR_RANK;
     PREV_RANK=NXT_RANK;                                            20
 END;
 RANK=NEW_RANK;
-/* WE NOW MUST ORDER THE DO END LISTS. */
 ALLOCATE DOTAB(#LOOPS);
 ALLOCATE ENDTAB(#LOOPS);
0/* MOVE THE DATA AND FREE THE LINKED LIST. */
 DO_END_PTR=START_PTR;
 DO ENT#=1 TO #LOOPS WHILE(DO_END_PTR¬=NULL);                     22
     LOC(ENT#)=DO_LOC;
     DO_LOOP_VAR(ENT#)=LOOP_VAR;
     ENDTAB(ENT#)=END_LOC;
     SAVE_PTR=NEXT_DO_END;
     FREE DO_END_LIST;
     DO_END_PTR=SAVE_PTR;
 END;
0/* SORT THE PRESUMABLY SHORT LISTS. */
 SWITCHES='1'B;
 DO J= #LOOPS-1 TO 1 BY -1 WHILE(SWITCHES);                        22
     SWITCHES='0'B;
     DO I=1 TO J;
          IF LOC(I)>LOC(I+1) THEN DO;
               TMP=LOC(I+1);
               T_LOOP=DO_LOOP_VAR(I+1);
               DOTAB(I+1)=DOTAB(I);
               LOC(I)=TMP;
               DO_LOOP_VAR(I)=T_LOOP;
               SWITCHES='1'B;
          END;
     END;
 END;
 SWITCHES='1'B;
 DO J=#LOOPS-1 TO 1 BY -1 WHILE(SWITCHES);
     SWITCHES='0'B;
     DO I=1 TO J;
          IF ENDTAB(I)>ENDTAB(I+1) THEN DO;
               TMP=ENDTAB(I+1);
               ENDTAB(I+1)=ENDTAB(I);
               ENDTAB(I)=TMP;
               SWITCHES='1'B;
          END;
     END;
 END;
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=GDCLT,EXTREF');
 GDCLT: PROC;
0/* 'GDCLT' TAKES THE STORAGE ENTRIES AS INPUT AND GENERATES A                1    2
 TABLE. THE TABLE CONTAINS THE NECESSARY INFORMATION FOR GENERATING           1    2
 DECLARATIONS IN A HIGH LEVEL LANGUAGE.  */                                   1    2
0           %INCLUDE INCLIB(DFILE);                                           1    2
            %INCLUDE INCLIB(DREPORT);                                         1    3
            %INCLUDE INCLIB(DFIELD);                                          1    4
            %INCLUDE INCLIB(DSEDIR);                                          1    5
            %INCLUDE INCLIB(DANY);                                            1    6
            %INCLUDE INCLIB(DTAPE);                                           1    7
                            .

            %INCLUDE INCLIB(DDISK);                                           1    8
            %INCLUDE INCLIB(DRECGRP);
            %DCL MAX_LEN_CNAME FIXED;
            %MAX_LEN_CNAME=32;
            %DCL MAX_#_MEMBERS FIXED;                                         1    9
            %MAX_#_MEMBERS=24;                                                1   10
  %DCL MAX#_INTERIMS FIXED;
  %MAX#_INTERIMS=64;
0           DCL 1 FIELD_DCL_PROTO BASED (P),                                  1   11
            /* THIS IS THE PROTOTYPE DECLARATION OF A FIELD OR INTERIM */
            2 TYPE CHAR(2),      /* 'FD' FOR FIELD OR 'IN' FOR INTERIM */
            2 FIELD_LEVEL FIXED (3) DEC,
            2 FIELD_NAME CHAR(LENGTH_KEY_NAME),                               1   11
            2 MAX_REPETITION FIXED DEC(3),
            2 FIELD_TYPE CHAR(1),                                             1   11
            2 FIELD_LEN_TYPE CHAR(1),                                         1   11
            2 MAX_LEN FIXED (5,0) DECIMAL,                                    1   11
            2 MIN_LEN FIXED (5,0) DECIMAL   ;                                 1   11
0           DCL 1 FILE_DCL_PROTO BASED (P),                                   1   12
             /* THIS IS THE PROTOTYPE DECLARATION OF A FILE OR A REPORT */    1   12
            2 TYPE CHAR(2),      /* 'FL' FOR FILE DECLARATION */
            2 FILE_LEVEL FIXED (3) DEC,
            2 FILE_NAME CHAR (LENGTH_KEY_NAME),                               1   12
            2 FILE_FLOW CHAR(1),                                              1   12
            2 FILE_ORG CHAR(1);                                               1   12
0           DCL 1 RECORD_DCL_PROTO BASED (P),                                 1   13
            /* THIS IS THE PROTOTYPE  DECLARATION OF  A RECORD OR A REPORT    1   13
            ENTRY */                                                          1   13
            2 TYPE CHAR(2),      /* 'RC' FOR RECORD STRING DECLARATION  */
            2 RECORD_LEVEL FIXED (3,0) DECIMAL,
            2 RECORD_NAME CHAR(MAX_LEN_CNAME),
            2 RECORD_LENGTH_TYPE CHAR(1),                                     1   13
            2 RECORD_LENGTH FIXED DEC(5,0),
            2 DEF_RECNAME CHAR(MAX_LEN_CNAME);
0           DCL 1 DCL_PROTO BASED (P),                                        1   14
            /* THIS GENERAL DECLARATION IS USED FOR STRUCTURES */             1   14
            2 TYPE CHAR(2),/* 'FP' FOR FILE NAME AT TOP LEVEL OF STRUCTURE;
                              'PR' FOR RECORD DCL AT 2ND LEVEL OF STRUCTURE
                              'GP' FOR GROUP DCL IN THE STRUCTURE  */
            2 LEVEL FIXED (3,0) DEC,
            2 NAME CHAR(LENGTH_KEY_NAME),                                     1   14
            2 MAX_REPETITION FIXED DEC(3);
 -DCL PINTR(MAX#_INTERIMS)  POINTER,  #PINTR FIXED BIN;
  /* ARRAY OF POINTER TO 'INTERIM' STORAGE ENTRIES AND # OF THEM */
 -          DCL PNODE CHAR (LENGTH_KEY_NAME)STATIC;
  /* 'PNODE' DENOTES  THE PARENT FILE */                                      1   16
0           DCL RETREVE ENTRY (CHAR(*),CHAR(*) VAR,POINTER, (*) POINTER,      1   17
            FIXED BINARY) EXTERNAL;                                           1   17
            DCL START POINTER EXT;
            DCL CHPTHLD ENTRY(CHAR(*))RETURNS(CHAR(MAX_LEN_CNAME)VAR);
            %DCL MAX_#_NAMES FIXED;                                           1   20
            %MAX # NAMES=24;                                                  1   21
```

```
0         DCL (SOURCE_ARRAY(MAX_#_NAMES),TARGET_ARRAY(MAX_#_NAMES),           1   22
          SOURCE_TARGET_ARRAY(MAX_#_NAMES),FILE_ARRAY(MAX_#_NAMES))
           POINTER;
 /* FILE_ARRAY CONTAINS THE POINTERS TO THE STORAGE_ENTRIES WHERE A          1   23
 FILE IS DEFINED */                                                          1   23
 /* SOURCE_ARRAY CONTAINS SOURCE ONLY FILES, TARGET_ ARRAY TARGET            1   23
  ONLY_FILES_AND_SOURCE_TARGET_ARRAY THE POINTERS TO THE                     1   23
 STORAGE_ENTRIES WHERE THE UPDATE FILES ARE DEFINED */                       1   23
0         DCL (#SOURCE_ARRAY,#TARGET_ARRAY,#SOURCE_TARGET_ARRAY,             1   23
               #FILE_ARRAY) FIXED BIN;
 /* #SOURCE_ARRAY IS THE NUMBER OF POINTERS INTHE SOURCE_ARRAY */            1   24


 /* #TARGET_ARRAY IS THE NUMBER OF POINTERS IN TARGET_ARRAY AND              1   24
 #SOURCE_TARGET_ARRAY IS THE NUMBER OF POINTERS IN SOURCE_TARGET_ARRAY       1   24
 #FILE_ARRAY IS THE NUMBER OF POINTERS IN FILE_ARRAY */                      1   24
0         DCL NULL BUILTIN;                                                  1   24
          DCL FILE_TITLE  CHAR(LENGTH_KEY_NAME);                             1   25
          DCL FORM_TREE ENTRY(POINTER,FIXED(2),FIXED(3));
 /* FORM_TREE IS A RECURSIVE PROCEDURE USED FOR  TRAVERSING THE TREE         1   27
  IN PREORDER */                                                             1   27
          %DCL MAX_LEN_INPUT FIXED;                                          1   27
          %MAX_LEN_INPUT=100;                                                1   28
          DCL STRING CHAR(MAX_LEN_INPUT) VAR ;                               1   29
          DCL STANDARD_NAME CHAR(LENGTH_KEY_NAME);                           1   30
 /* STRING IS THE STRING THAT IS USUALLY PASSED TO THE PROCEDURE
 RETREVE AS THE FIRST PARAMETER . STANDARD_NAME DENOTES A STANDARD NAME      1   31
 USED  IN BUILDING THE STORAGE_ENTRIES */                                    1   31
0DCL FILETYPE CHAR(LENGTH_KEY_NAME) INIT('$FILE') STATIC;
 DCL REPTTYPE CHAR(LENGTH_KEY_NAME) INIT('$REPT') STATIC;
 /* FILETYPE AND REPTTYPE ARE USED IN RETRIEVALS */
0DCL OR CHAR(1) STATIC INIT('|'), ANDNOT CHAR(2) STATIC INIT('&¬');
 /* LOGICAL OPERATORS USED IN RETRIEVALS  */
0         DCL TEMP_RECORD_NAME CHAR(LENGTH_KEY_NAME);
          DCL TEMP_FILE_NAME CHAR(LENGTH_KEY_NAME);
             DCL STORAGE_NAME CHAR(LENGTH_KEY_NAME);
 /* TEMPORARY NAMES FOR RETRIEVED ITEMS OF LENGTH 'LENGTH_KEY_NAME' */
0         DCL SAVE_STORAGE_PTR POINTER;
 /* SAVES POINTER TO STORAGE ENTRY OF STORAGE MEDIUM DESCRIPTION */
0         DCL FILEST ENTRY(CHAR(*))RETURNS(CHAR(2));
          DCL FILE_ST_TYPE CHAR(2);
-         STANDARD_NAME='$FILE';                                             1   31
          STRING=STANDARD_NAME||'|';                                         1   32
          STANDARD_NAME='$REPT';
          STRING=STRING||STANDARD_NAME;                                      1   34
          CALL RETREVE(STRING,'',START,FILE_ARRAY,#FILE_ARRAY);              1   35
 /* GET ALL THE FILES AND REPORTS */                                        1   36
              #SOURCE_ARRAY,#TARGET_ARRAY,#SOURCE_TARGET_ARRAY=0;          1   36
 -            DO I=1 TO #FILE_ARRAY;
0/* THEPOINTERS TO THE STORAGE ENTRIES WHERE THE SOURCE FILES ARE         2   38
 DEFINED  ARE SET IN "SOURCE_ARRAY" INDEXED BY "#SOURCE_ARRAY" ,           2   38
 THE POINTERS TO THE STORAGE ENTRIES WHERE THE TARGET FILES ARE DEFINED    2   38
 ARE SET IN "TARGET_ARRAY" INDEXED BY "#TARGET_ARRAY" AND THE POINTERS     2   38
 TO THE STORAGE ENTRIES OF THE FILES WHICH ARE BOTH SOURCE AND TARGET      2   38
 IN "SOURCE_TARGET_ARRAY" INDEXED BY "#SOURCE_TARGET_ARRAY" */             2   38
0             STORAGE_PTR=FILE_ARRAY(I);                                   2   38
              FILE_TITLE=STORAGE_ENTRY.NAME(I);                            2   39
              FILE_ST_TYPE=FILEST(FILE_TITLE);
              /*FILEST RETURNS 'SR' IF THE FILE IS SOURCE ONLY;
              'TG' IF THE FILE IS TARGET ONLY; 'ST' IF THE FILE IS BOTH
              SOURCE AND TARGET*/                                                    2
0                IF FILE_ST_TYPE='TG' THEN
                    DO;                                                    4   47
 /* IT IS A TARGET ONLY FILE */                                            4   48
                    #TARGET_ARRAY=#TARGET_ARRAY+1;                         4   48
                    TARGET_ARRAY(#TARGET_ARRAY)=FILE_ARRAY(I);             4   49
                    END;                                                   4   50
0                ELSE
0                   IF FILE_ST_TYPE='SR' THEN
                       DO;                                                 5   52
 /* IT IS A SOURCE ONLY FILE */                                            5   53
                       #SOURCE_ARRAY=#SOURCE_ARRAY+1;                      5   53  1
                       SOURCE_ARRAY(#SOURCE_ARRAY)=FILE_ARRAY(I);          5   54
                       END;                                                5   55
0                   ELSE                                                   4   56
0                   IF FILE_ST_TYPE='ST' THEN


                          DO;                                              6   57 3
 /* IT IS A SOURCE AND TARGET FILE */                                      6   58
                          #SOURCE_TARGET_ARRAY=#SOURCE_TARGET_ARRAY+1;     6   58
                          SOURCE_TARGET_ARRAY(#SOURCE_TARGET_ARRAY)=FILE_A 6   59
 RRAY(I);                                                                  6   59
                          END;                                             6   60
0                      ELSE  CALL SYSERR('DCCLT:'||
                     FILE_TITLE||' NEITHER SOURCE NOR TARGET FILE');
              END;                                                         2   62
 -            DO I=1  TO #SOURCE_ARRAY;                                                4
 /* FOR EVERY SOURCE-ONLY FILE, GENERATE THE DECLARATION OF THE FILE       2    6
 AND OF ITS RECORD */                                                      2   64
              STORAGE_PTR=SOURCE_ARRAY(I);                                 2   64
              LOCATE FILE_DCL_PROTO FILE(DCLTAB);                          2   65
              FILE_DCL_PROTO.FILE_NAME=CHPTPLR(STORAGE_ENTRY.NAME(I))||
                  'S';
              FILE_DCL_PROTO.FILE_FLOW='I';
               /*FILL UP FILE DCL TABLE*/
              CALL ENTER_FILE_INFO;
 /* THE PROCEDURE FREC GENERATES THE RECORD STRING DECLARATION TABLE
 IF THE ARGUMENT IS 'I' THEN IT IS INPUT , IF NOT , OUTPUT */              2   74
              FILE_ST_TYPE='SR';
              CALL FREC(I);                                                2   74
              END;                                                         2   75  5
```

```
  DCL      1 DOTAB    (*) CTL EXT,
                2 LOC FIXED BIN,
                2 DO_LOOP_VAR CHAR(EACH_NAME_LEN);
  DCL ENDTAB    (*) FIXED BIN CTL EXT;
  DCL #LOOPS FIXED BIN EXT;
 0/* IMPORTANT LOCAL VARIABLES USED BY INTERNAL SUBROUTINES. */
  /* DO_END_LIST IS THE TEMPORARY REPOSITORY OF THE BOUNDS AND
  ITERATION VARIABLES FOR THE LOOPS. IT IS A LINKED LIST. THE TOP ELT
  IS ALWAYS REFERENCED BY 'START_PTR'. AT THE END OF THE MAIN
  ROUTINE THE LIST WILL BE CONVERTED INTO ARRAYS AND SORTED BY
  LOCATION IN THE ORDER VECTOR. */
  DCL       1 DO_END_LIST BASED(DO_END_PTR),
                2 DO_LOC FIXED BIN,
                2 END_LOC FIXED BIN,
                2 LOOP_VAR CHAR(EACH_NAME_LEN),
                2 NEXT_DO_END PTR;
  DCL START_PTR PTR;
  /* NAME_LIST IS THE LIST OF FOREACH NAMES WHICH START LOOPS AT
  THIS POINT. */
  DCL        1 NAME_LIST BASED(NAME_PTR),
                 2 EACH_NAME CHAR(EACH_NAME_LEN),
                 2 NEXT_NAME PTR;
  DCL EACH_PTR PTR;
 0/* OTHER VARIABLES. */
  DCL T_LOOP CHAR(EACH_NAME_LEN);
  DCL (T_LIST_PTR , SAVE_PTR) PTR;
  DCL (YES_DEP(DICTIND), NOT_DEP(DICTIND))FIXED BIN;
  DCL SAVE_ORDER_DEP(DICTIND) FIXED BIN;
  DCL (VEC_ENTRY#, NODE#, PART_NODE#, #YES, #NOT, MOVE#) FIXED BIN;
  DCL LAST_DEP FIXED BIN;
  DCL (ENT#, I, J, TMP) FIXED BIN;
  DCL (DISP, #DEPS) FIXED BIN;
  DCL SWITCHES BIT(1);
  DCL NULL  BUILTIN;
  DCL NODE_TYPE CHAR(4);
  DCL REC_PTR PTR,
      1 MOVE_REC_FLD BASED(MOVE_PTR),
         2 FILE_NAME CHAR(     LEN_DICT_ENTRY),
         2 REC_NAME CHAR(     LEN_DICT_ENTRY),
         2 NEXT_PAIR PTR;
  DCL TOP_PTR PTR;
  DCL #MOVED_ORDER FIXED BIN;
  DCL (#MOVED,NEXT_LOC, MOVE_NODE(DICTIND), MOVED_ORDER(DICTIND))
              FIXED BIN ;
  DCL RANK(*) FIXED BIN EXT CTL;
  DCL NEW_RANK(DICTIND) FIXED BIN;
  DCL (CUR_ORDER,CUR_RANK,PREV_RANK,NXT_RANK) FIXED BIN;
```

```
0              DO I=1 TO #TARGET_ARRAY;
 /* THE SAME THING IS DONE FOR TARGET ONLY FILES */                         2   77
               STORAGE_PTR=TARGET_ARRAY(I);                                 2   77
               LOCATE FILE_DCL_PROTO FILE(DCLTAB);                          2   78
               FILE_DCL_PROTO.FILE_NAME=CHRTRLB(STORAGE_ENTRY.NAME(1))||
               'T';
               FILE_DCL_PROTO.FILE_FLOW='C';
                /*FILL UP FILE DCL TABLE*/
               CALL ENTER_FILE_INFO;
               FILE_ST_TYPE='TG';
               CALL FREC(2);                                                2   87
               END;                                                         2   88
 /*GENERATE DECLARATION FOR INPUT_OUTPUT FILES*/
0              DO I = 1 TO #SOURCE_TARGET_ARRAY;                                        6
               STORAGE_PTR=SOURCE_TARGET_ARRAY(I);                          2   90
               LOCATE FILE_DCL_PROTO FILE(DCLTAB);                          2   91
                  FILE_DCL_PROTO.FILE_NAME=CHRTRLB(STORAGE_ENTRY.NAME(1))||
         'U';
               FILE_DCL_PROTO.FILE_FLOW='U';
                /*FILL UP FILE DCL TABLE*/
                  CALL ENTER_FILE_INFO;
               FILE_ST_TYPE='ST';
 /* FOR AN UPDATE FILE TWO RECORDS ARE GENERATED */                         2  100
               CALL FREC(1);                                                2  100
               CALL FREC(2);                                                2  101
               END;                                                         2  102
-              DO I=1 TO #SOURCE_ARRAY;
 /* GENERATING THE TABLES FOR THE STRUCTURE WHICH HAS THE FILE              2  104
 AT LEVEL ONE */                                                            2  104
               STORAGE_PTR=SOURCE_ARRAY(I);                                 2  104
               PNODE='$P'|| STORAGE_ENTRY.NAME(1);
 /* THE PROCEDURE 'FORM_TREE' TRAVERSES THE TREE GENERATED BY THE           2  106
 FILE STRUCTURE IN PREORDER */                                              2  106
               CALL FORM_TREE(SOURCE_ARRAY(I),1,0);
               END;                                                         2  107
0              DO I=1 TO #TARGET_ARRAY;
               STORAGE_PTR=TARGET_ARRAY(I);                                 2  109
 /* 'PNODE' IS A GLOBAL PARAMETER FOR THE PROCEDURE 'FORM_TREE' */          2  110
 /* IT IS NECESSARY FOR THE 'RETREVE' PROCEDURE */                          2  110


               PNODE='$P'||        STORAGE_ENTRY.NAME(1);
               CALL FORM_TREE(TARGET_ARRAY(I),1,0);
               END;                                                         2  112
0              DO I=1 TO  #SOURCE_TARGET_ARRAY;
               STORAGE_PTR=SOURCE_TARGET_ARRAY(I);                          2  114
               PNODE='$P'||        STORAGE_ENTRY.NAME(1);
             /* FOR ST FILES, FORM THE TREE STARTING AT LEVEL 2 BECAUSE
                THE 'FLD.' WILL FALL AT LEVEL 1  */
               CALL FORM_TREE(SOURCE_TARGET_ARRAY(I),2,0);
               END;                                                         2  117
-     /* FIND INTERIM NAMES  */
0     STANDARD_NAME='$INTR';
      CALL RETREVE(STANDARD_NAME,'',START,PINTR,#PINTR);
      DO I=1 TO #PINTR;
               STORAGE_PTR=PINTR(I);
               DP=DATA_PT;
                    LOCATE FIELD_DCL_PROTO FILE(DCLTAB);
                    FIELD_NAME=STORAGE_ENTRY.NAME(1);
                    FIELD_DCL_PROTO.TYPE='IN';
                    FIELD_LEVEL=2;
                       IF FIELD.FIELD_TYPE=0 THEN
                       FIELD_DCL_PROTO.FIELD_TYPE='C';
                       ELSE
                          IF FIELD.FIELD_TYPE=1 THEN
                          FIELD_DCL_PROTO.FIELD_TYPE='B';
 ELSE IF  FIELD.FIELD_TYPE=2 THEN FIELD_DCL_PROTO.FIELD_TYPE='N';
 ELSE FIELD_DCL_PROTO.FIELD_TYPE='F';
                       IF FIELD.FIELD_LEN_TYPE=0 THEN
                       FIELD_DCL_PROTO.FIELD_LEN_TYPE='F';
                       ELSE
                       FIELD_DCL_PROTO.FIELD_LEN_TYPE='V';
                    MAX_LEN=FIELD.FIELD_LEN_MAX;
                    MIN_LEN=FIELD.FIELD_LEN_MIN;
                   FIELD_DCL_PROTO.MAX_REPETITION=0;
      END;
0          CLOSE FILE(DCLTAB);
```

```
1FREC:        PROC(IOTYPE);
0/* THIS ROUTINE FORMS THE RECORD STRING DECLARATION TABLE*/
0             DCL IOTYPE FIXED DEC(1);
 /*1=INPUT,2=OUTPUT*/
0             DCL DEVICE(2)CHAR(7)INIT('CARDS','PRINTER');
-              TEMP_FILE_NAME=STORAGE_ENTRY.NAME(1);
              TEMP_RECORD_NAME=STORAGE_ENTRY.NAME(2);
-             LOCATE RECORD_DCL_PROTO FILE(DCLTAB);                          2  125
              RECORD_DCL_PROTO.RECORD_NAME=CHPTRLB(TEMP_RECORD_NAME)||
               '_S';
0             IF FILE_ST_TYPE='ST' THEN
               /*FILE BOTH SOURCE AND TARGET*/
               /*MODIFY RECORD STRING NAME WITH PREFIX 'OLD_' OR 'NEW_'*/
              DO;
                 IF IOTYPE=1 THEN
                 RECORD_DCL_PROTO.RECORD_NAME='OLD_'||
                 RECORD_DCL_PROTO.RECORD_NAME;
                 ELSE
                 RECORD_DCL_PROTO.RECORD_NAME='NEW_'||
                 RECORD_DCL_PROTO.RECORD_NAME;
              END;
0             RECORD_DCL_PROTO.TYPE='RC';                                    2  128
              RECORD_DCL_PROTO.RECORD_LEVEL=0;                               2  129
0               IF STORAGE_NAME=' ' THEN
                /*NO STORAGE NAME SPECIFIED;
                SET DEFAULT RECORD LENGTHS*/
                    DO;                                                      4  132


                    RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='F';                 4  133
                       IF IOTYPE=1 THEN
                       RECORD_DCL_PROTO.RECORD_LENGTH=80;                    5  135
                       ELSE                                                  5  136
                       RECORD_DCL_PROTO.RECORD_LENGTH=120;                   5  136
              CALL PWARN('INCOMPLETENESS):STORAGE MEDIUM FOR FILE '||
               TEMP_FILE_NAME||' MISSING; '||DEVICE(IOTYPE)||' ASSUMED');
                    END;                                                     4  138
```

```
0            ELSE                                                       3  139
               DO;                                                      4  139
                  /* SET DP (DATA TABLE POINTER) TO THE STORAGE
                    MEDIUM DESCRIPTION STORAGE ENTRY THAT WAS SAVED */
                  DP=SAVE_STORAGE_PTR->DATA_PT;
                   IF ANY_STMT.TYPE='CARD'|ANY_STMT.TYPE='PNCH' THEN    5  141
                     DO;                                                6  142
                     RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='F';           6  143
                     RECORD_DCL_PROTO.RECORD_LENGTH=80;                 6  144
                     END;                                               6  145
                   IF ANY_STMT.TYPE='DISK' |ANY_STMT.TYPE='TAPE'|
                    ANY_STMT.TYPE='TERM' THEN
                     DO;                                                6  147
                        IF DISK.RECFM<2 THEN
                           DO;                                          8  149
                           RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='F';     8  150
                           RECORD_DCL_PROTO.RECORD_LENGTH=DISK.LRECL;   8  151
                           END;                                         8  152
                        ELSE
                           DO;                                          8  154
                           RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='V';     8  155
                           RECORD_DCL_PROTO.RECORD_LENGTH=DISK.LRECL;   8  156
                           END;                                         8  157
                     END;                                               6  158
                   IF ANY_STMT.TYPE='PRNT' THEN                         5  159
                     DO;                                                6  160
                     RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='F';           6  161
                     RECORD_DCL_PROTO.RECORD_LENGTH=120;                6  162
                     END;                                               6  163
               END;                                                     4  177
0          IF RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='F' THEN
           DO;
         /*FIND RECORD NAME OVER WHICH RECORD STRING WILL BE OVERLAID*/
             IF FILE_ST_TYPE='ST' THEN
             DO;
               IF IOTYPE=1 THEN RECORD_DCL_PROTO.DEF_RECNAME='OLD.'||
 CHPTRLB(TEMP_FILE_NAME)|| '.'||
               TEMP_RECORD_NAME;
               ELSE
                              RECORD_DCL_PROTO.DEF_RECNAME='NEW.'||
 CHPTRLB(TEMP_FILE_NAME)|| '.'||
               TEMP_RECORD_NAME;
             END;
             ELSE RECORD_DCL_PROTO.DEF_RECNAME=
 CHPTRLB(TEMP_FILE_NAME)||'.'||TEMP_RECORD_NAME;
            END;
            ELSE RECORD_DCL_PROTO.DEF_RECNAME=' ';
-          END FREC;
```

```
1FORM_TREE: PROC(DEF_S_E,T_LEVEL,MAX_REP)  RECURSIVE;
0/* THIS PROCEDURE GENERATES TABLE ENTRIES REPRESENTING THE HIERARCHICAL
    RECORD_STRUCTURE */
 0             DCL SAVE_PTR POINTER,                                           2  185
               /*  'SAVE_PTR ' IS USED FOR SAVING THE STORAGE_PTR IN           2  185
               HE PROCEDURE FORM_TREE */                                       2  185
               DEF_S_E POINTER,                                                2  185


               /* DEF_S_E IS THE POINTER WHERE NODE WHICH IS THE ROOT OF TH    2  185
 E             SUBTREE TO BE TRAVERSED IS DEFINED */                           2  185
               T_LEVEL FIXED DEC(3);
               /* T_LEVEL IS THE LEVEL OF THE NODE IN THE TREE */              2  185
       DCL MAX_REP FIXED DECIMAL (3);
       /*MAX_REP IS MAXIMUM NUMBER OF REPETITIONS OF GROUP OR FIELD*/
       DCL TEMP_MAX_REP FIXED DEC(3);
          DCL II FIXED BIN;
          DCL (JJ,MEM#)  FIXED BIN STATIC;
          DCL NODE CHAR(LENGTH_KEY_NAME);
   /* NODE IS THE NAME OF THE NODE */                                          2  186
               DCL STACK(MAX_#_MEMBERS) POINTER;                               2  186
   /* 'STACK' STORES THE STORAGE POINTERS OF THE DESCENDENTS */                2  187
               DCL #STACK FIXED BINARY;                                        2  187
 -             STORAGE_PTR=DEF_S_E;                                            2  188
               DP=DATA_PT;                                                     2  189   1
 -               IF ANY_STMT.TYPE='FLD ' THEN                                  3  190
               DO;                                                             4  191
   /* IF THE STRUCTURE IS A FIELD THEN FILL IN THE INFO ON ITS DCL */
                     LOCATE FIELD_DCL_PROTO FILE(DCLTAB);                      4  192
                     FIELD_NAME=STORAGE_ENTRY.NAME(1);                         4  193
                     FIELD_DCL_PROTO.TYPE='FD';                                4  194
                     FIELD_LEVEL=T_LEVEL;                                      4  195
                        IF FIELD.FIELD_TYPE=0 THEN
                        FIELD_DCL_PROTO.FIELD_TYPE='C';
                        ELSE                                                   5  198
                           IF FIELD.FIELD_TYPE=1 THEN
                           FIELD_DCL_PROTO.FIELD_TYPE='9';
   ELSE IF  FIELD.FIELD_TYPE=2 THEN FIELD_DCL_PROTO.FIELD_TYPE='N';
   ELSE FIELD_DCL_PROTO.FIELD_TYPE='F';
                        IF FIELD.FIELD_LEN_TYPE=0 THEN                         5  200
                        FIELD_DCL_PROTO.FIELD_LEN_TYPE='F';                    5  201
                        ELSE                                                   5  202
                        FIELD_DCL_PROTO.FIELD_LEN_TYPE='V';
                     MAX_LEN=FIELD.FIELD_LEN_MAX;                              4  203
                     MIN_LEN=FIELD.FIELD_LEN_MIN;                              4  204
                    FIELD_DCL_PROTO.MAX_REPETITION=MAX_REP;
                     END;                                                      4  205
 -                ELSE                                                         3  206
 0                   DO;                                                       4  206   2
   /* IF THE STRUCTURE IS NOT A FIELD THEN TRAVERSE FURTHER IN                 4  207
   PREORDER */       LOCATE DCL_PROTO FILE(DCLTAB);                            4  207
                     DCL_PROTO.LEVEL=T_LEVEL;                                  4  208
                     DCL_PROTO.NAME=STORAGE_ENTRY.NAME(1);                     4  209
                     DCL_PROTO.MAX_REPETITION=MAX_REP;
                        IF ANY_STMT.TYPE='FILE'|ANY_STMT.TYPE='REPT' THEN      5  210
                          DO;                                                  6  211
                          CALL RETREVE(STORAGE_ENTRY.NAME(2)||'&'||PNODE,'     6  212   3
   RECD',START,STACK,#STACK);
                             IF #STACK=0 THEN
                             CALL RETREVE(STORAGE_ENTRY.NAME(2)||'&'||PNOD     7  214
   E,'RPTN',START,STACK,#STACK);
                          DCL_PROTO.TYPE='FR';                                 6  216
                          CALL FORM_TREE(STACK(1),T_LEVEL+1,0);                         4
                          RETURN;
                          END;                                                 6  218
                        IF ANY_STMT.TYPE='RECD'|ANY_STMT.TYPE='RPTN' THEN      5  219
                        DCL_PROTO.TYPE='RR';                                   5  220
                        ELSE                                                   5  221
                        DCL_PROTO.TYPE='GP';                                   5  221
                     SAVE_PTR=STORAGE_PTR;                                     4  222
                        DO II=2 TO #KEYS-2;
                        /* FOR EACH DESCENDANT*/
 /* THERE ARE #KEYS-3 SUBTREES */                                           5  224
                     NCDE=STORAGE_ENTRY.NAME(II);                           5  224
                     /*FIND OUT HOW MANY TIMES THE MEMBER REPEATS*/
                     MEM#=II-1; /* THE MEMBER# IS 1 LESS THAN II */
                     IF RECGRP.#SUB(MEM#)=0 THEN TEMP_MAX_REP=C;
                     ELSE IF RECGRP.#SUB(MEM#)=1 THEN TEMP_MAX_REP=
                             RECGRP.FIRST_SUB(MEM#);
                    ELSE
                       DO;
                       /*THERE ARF 2 SUBSCRIPTS: IF THE UPPER IS 1
                       (EC(0:1)) THEN FIELD IS REALLY JUST OPTIONAL;
                       THERFFORE PRETENC IT  COES NCT REPEAT*/
                       IF RECGRP.SECCNC_SUB(MEM#)=1 THEN TEMP_MAX_REP=0;
                    FLSE
                    TEMP_MAX_REP=RECGRP.SECCNC_SUB(MEM#);
                    END;
                     CALL RETREVE(NCCE||'&'||PNCDE,'',START,STACK,
                     #STACK);
 /* FIND WHERE THE NCDE IS DEFINEC */                                       6  230
                     JJ=1;
                     STORAGE_PTR=STACK(1);
                     DO WHILE(STORAGE_ENTRY.NAME(1)¬=NODE &
                        JJ<=#STACK);
                        JJ=JJ+1;
                        STUPAGE_PTR=STACK(JJ);
                     END;
                        IF JJ>#STACK THEN CALL SYSERR
                         ('GDCLT: MEM DEF NCT FCUND');
                     CALL FORM_TRCE(STACK(JJ),T_LEVEL+1,TEMP_MAX_REP);
                     STORAGE_PTR=SAVE_PTR;                                  5  233
                     DP=DATA_PT;
                     END;                                                   5  234
                  END;                                                      4  235
-           END FORM_TREE;                                                  2  236
1ENTER_FILE_INFO:  PROC;
0/* THIS PROCEDURE ENTERS THE REMAINING INFCRMATICN INTO THE FILE
      DECLARATICN TABLE  */
0DCL PMED(MAX_#_NAMES) PCINTER, #PMEC FIXEC BIN;
 /* PMED IS ARRAY OF POINTERS USEC BELOW IN RETRIEVING  MEDIUM
  DESCRIPTION STORAGE ENTRIES:  #PMEC IS THE NUMBER OF SUCH
          ENTRIES RETRIEVEC  */
            FILE_DCL_PROTC.TYPE='FL';
            FILE_CCL_PROTC.FILE_LEVEL=0;
            /*DETERMINE FILE ORGANIZATICN*/
            STORAGE_NAME=STCRAGE_ENTRY.NAME(3);
            IF STORAGE_NAME=' ' THEN
            CO;
               FILE_CCL_PPCTC.FILE_CRG='S';
               RETURN;
            END;
            ELSE
                     /*RETREVE MEDIUM DESCRIPTION ENTRY, FOR INFO ON
                          FILE_ORGANIZATICN  */
            CO;
                  CALL RETREVE(STORAGE_NAME||ANDNCT||FILETYPE||
                              ANDNCT||FEPTTYPE,'',START,PMED,#PMED);
                  IF #PMEC=0  THEN
                  CALL SYSERR
                    ('GDCLT:NO MEDIUM DESC FCR '||STORAGE_NAME);
           SAVE_STORAGE_PTR=PMED (1);
           DP=SAVE_STORAGE_PTR->DATA_PT;
            IF ANY_STMT.TYPE='DISK' THEN FILE_DCL_PRCTC.FILE_ORG=
            DISK.CRGANIZATICN;


            FLSE FILF_DCL_PROTC.FILE_CRG='S';
            END;
0END ENTER_FILE_INFO;
-        END GDCLT;
```

```
*PRCCESS('NST,MACPC,N=GENCCND,EXTREF');
 /* THIS PRCCEDURE GENERATES CCDE FOR CONDITICNAL*/
 %CCL MAX_LEN_QNAME FIXEC;
 %MAX_LEN_QNAME=32;
 GENCOND: PROC(FLOW_PTR);
 DCL FLCW_PTP POINTER;
      DCL CHPTPLB ENTRY(CHAP(*)) RETURNS(CHAR(MAX_LEN_CNAME)VAR);
 DCL 1 FLOWTAB_COND BASED(P),
           2 NCDE# FIXED BIN,
           2 NODE_TYPE CHAP(4), /*CCND*/
           2 CCND_NAME CHAR(MAX_LEN_CNAME);
 DCL FLP PCINTER;
 P=FLCW_PTR;
 CALL WRPLI('IF '||CHPTPLB(COND_NAME)||' THEN','PL1EX');
 END;

*PRCCESS('NST,MACPO,N=GENCC');
 GENCC: PROC(FLOW_PTR);
 /*THIS PROCEDURE GENERATES THE DC STATEMENT*/
 %CCL MAX_LEN_FOREACH FIXED;
 %MAX_LEN_FOREACH=18;
 %CCL MAX_LEN_QNAME FIXED;
 %MAX_LEN_QNAME=32;
      DCL FLCW_PTP POINTER;
      DCL 1 FLCWTAB_DC BASED(P),
           2 NCDE# FIXED BIN,
           2 TYPE CHAR(4),
           2 FOREACH_NAME CHAR(MAX_LEN_FCREACH),
           2 UPPER_TYPE CHAP(1),
           2 UPPER# FIXED DEC,
           2 UPPER_NAME CHAP(MAX_LEN_QNAME);
      DCL CHPTPLB ENTRY(CHAR(*)) RETURNS(CHAP(MAX_LEN_QNAME)VAR);
      P=FLOW_PTP;
      DCL TEMP_UPPER CHAP(MAX_LEN_CNAME)VAR;
 IF UPPER_TYPE='F' THEN
      PUT STRING(TEMP_UPPER) EDIT(UPPER#)(F(3));
 ELSE TEMP_UPPER=UPPER_NAME;
 CALL WRPLI('DO '||CHPTPLP(FOREACH_NAME)||'=1 TO '||
 TEMP_UPPER||';','PL1EX');
 END;
```

```
*PROCESS('NST,N=GENEND,MACRO');
 GENEND: PROC(FLOW_PTR);
 /*THIS PROCEDURE GENERATES THE 'END' STATEMENT GIVEN BY THE FLOWTAB
 ENTRY POINTED TO BY FLOW_PTR */
 %DCL MAX_LEN_NAME FIXED;
 %MAX_LEN_NAME=10;
 DCL FLOW_PTR POINTER;
 DCL 1 FLOWTAB_PROTO BASED(P),
       2 NODE# FIXED BIN,
       2 TYPE CHAR(4);
 P=FLOW_PTR;
 IF NODE#=0 THEN
 /* 'END' IF FOR END OF GENERATED MODULE; THEREFORE IT GOES AT END OF
 'PLIPROC' FILE */
      CALL
 WRPLI('END;','PLIPROC');
 ELSE /* END IS FOR A MATCHING 'DO';
 THEREFORE, IT GOES IN PLIEX_FILE*/


      CALL
 WRPLI('END;','PLIEX');
 END GENEND;

*PROCESS('NST,N=GENGOTO,MACRO');
 GENGOTO: PROC(FLOW_PTR);
 /*THIS PROCEDURE GENERATES A GOTO STATEMENT LABEL INDICATED IN "GOTO"
 FLOWCHART ENTRY, WHICH IS POINTED TO BY 'FLOW_PTR'*/
 %DCL MAX_LEN_LABEL  FIXED;
 %MAX_LEN_LABEL=14;
      DCL CHPTRLB  ENTRY(CHAR(*)) RETURNS(CHAR(32)VAR);
 DCL 1 FLOWTAB_PROTO BASED(P),
       2 NODE# FIXED BIN,
       2 TYPE CHAR(4),
       2 LABEL CHAR(MAX_LEN_LABEL);
 DCL FLOW_PTR POINTER;
 P=FLOW_PTR;
  CALL WRPLI('GO TO '||CHPTRLB(LABEL)||';','PLIEX');
 END;

*PROCESS('NST,N=GENLAB,MACRO');
 GENLAB:PROC(FLOW_PTR);
 /*THIS PROCEDURE GENERATES THE LABEL INDICATED BY THE FLOWTAB ENTRY
 POINTED TO BY 'FLOW_PTR' */
 %DCL MAX_LEN_LABEL  FIXED;
 %MAX_LEN_LABEL=14;
 DCL 1 FLOWTAB_PROTO BASED(P),
       2 NODE# FIXED BIN,
       2 TYPE CHAR(4),
       2 LABEL CHAR(MAX_LEN_LABEL);
 DCL FLOW_PTR POINTER;
      DCL CHPTRLB  ENTRY(CHAR(*)) RETURNS(CHAR(32)VAR);
 P=FLOW_PTR;
 CALL WRPLI(CHPTRLB(LABEL)||':;','PLIEX');
 END;
```

```
*PROCESS('NST,MACRC,N=GENFLT,SM=(2,72,1)');
 GENFLT:PRCC;
 /*THIS PROCEDURE CALLS RCUTINES TC GENERATE THE FLCWCHART TABLE*/
 /* WE USE THE ORDER VECTCR AND DICTYPE TC CALL THE APPROPRIATE
  RCUTINE WHICH WILL GENERATE THE CORRESPCNCING FLCWCHART RECORC. */
 %INCLUDE INCLIB(DDICT);
 DCL CRCER(DICTIND) FIXED BIN EXT CTL,
 /*VECTOR OF CRDER OF NODES*/
 J FIXED BIN, DICT# FIXED BIN, TYPE CHAR(4);
0DCL 1 FLOWTAB_DUMMY BASED(FLOWTAB_PTR),
        2 NCDE# FIXED BIN,
        2 NCDE_TYPE CHAR(4),
        2 NODE_NAME  CHAR(LEN_DICT_ENTRY);
 DCL #CONCAS FIXEC BIN EXT;
 /*CCUNTER FOR # OF ASSERTIONS THAT ARE CCNCITIONAL*/
0/* WE SIMPLY EXAMINE EACH ENTRY IN THE VECTOR ANC CALL ACCORDING TO
 ITS TYPE. */
 #CONCAS=C;
 /*INITIALLY, NO NODE(ASSERTION USES A CCNDITIONALLY_COMPLETED
 FUNCTION*/
-LOCP: DC J=1 TO DICTIND;                                                1
0     DICT#=ORDER(J);
      TYPE=DICTYPE(DICT#);
-/*CHECK DO_TABLE FCR POSSIBLE GENERATICN CF A DO*/
      CALL CHECKDO(J);                                                   2
0     /*CHECK CCND_TABLE FCR POSSIBLE CNEPATICN OF  CODE FOR CCNDITIONAL
      COMPLETION OF ASSERTICN*/
      CALL CHECONC(DICT#);                                               3
-/* CHECK THE "LABEL" TABLE FCR PCSSIBLE GENERATICN OF A LABEL  */
0      CALL CHECLAB(DICT#);                                              4
-/*CALL APPROPRIATE SUBRCUTINE TO GENERATE A FLCWCHART ENTRY FCR THIS
 TYPE*/
-     IF TYPE='ASTC' THEN CALL IDASSN(DICT#);                            5
         ELSE
 CASES: DO;
      IF TYPE='RECD' THEN CALL IDICCD(DICT#);
         ELSE DO;
      IF TYPE='FLO ' THEN CALL IDFLCAS(DICT#);
         ELSE DO;
      IF TYPE='INTR' THEN CALL IDFLCAS(DICT#);
         ELSF DO;
      IF TYPE= 'RPTN' THEN CALL IDICCC(DICT#);
         ELSE DO;
      IF TYPE='MODL' THEN CALL IDMCCNM(DICT#);
         ELSE DO;
-      /* GENERATE CUMMY FLOWCHART ENTRY FCR ALL CTHER CASES  */
       LOCATE FLOWTAB_CUMMY FILE(FLOWTAB);
        NCDE#=DICT#;
        NCDE_TYPE=TYPE;
        NODE_NAME=DICT(DICT#);
 END CASES;
0/*CHECK FCR GENERATION OF END CF LCCP*/
 CALL CHECEND(J);                                                       6
 END LCCP;                                                              7, 8
0/*GENERATE CODE FOR END CF PROGRAM*/
 CALL IDRSET;                                                           9
 CALL IDGCTO;
 CALL IDFIN;


 CALL IDENC:
0CLOSE FILE(FLOWTAB);
 /* THE FLOWCHART TABLE (FILE 'FLCWTAB') IS CCMPLETED;
     CLOSE IT AS CUTPUT, SO THAT SUBSECUENT RCUTINES CAN CPEN IT
     FOR INPUT.   */
-END GENFLT;
```

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=GENICCD');
 GENICCD:  PROC(FLOWTAB_REC_PTR);
 %DCL MAX_DEPTH_STACK FIXED;
 %MAX_DEPTH_STACK=27;
 %INCLUDE INCLIB(CDICT);
 %DCL LENGTH_KEY_NAME FIXED;
 %LENGTH_KEY_NAME=10;
 DCL FLOWTAB_REC_PTR POINTER;
0/* THE STRUCTURE "FTR" IS THE SAME AS "FLOWTAB_REC" IN THE ORIGINAL  */
 /* SPECIFICATIONS.                                                   */
 DCL 1 FTR BASED(FTR_PTR),
        2 NODE# FIXED BIN,
        2 NODE_TYPE CHAR(4), /* 'RECD' */
        2 RECNAME CHAR(LEN_DICT_ENTRY),
        2 IOMODE CHAR(2),
                          /* RD FOR READ
                             WR FOR WRITE
                             RW FOR REWRITE */
        2 FILENAME CHAR(MAX_L_NAME),
        2 ORG CHAR(1),
                          /* S = SEQUENTIAL
                             I = INDEXED     */
        2 KEYED FIXED BIN,
                             /* 0 = NOT KEYED
                                1 = KEYED     */
        2 KEYNAME               CHAR(MAX_L_NAME),
        2 PACKED                BIN FIXED(15),
                                                       /* 0 = NOT PACKED
                                                          1 = PACKED      */
        2 RECORD_ARITY          BIN FIXED(15),
        2 #SUBSTRUCTURES        BIN FIXED(15),
        2 SUBSTRUCTURE(N REFER(FTR.#SUBSTRUCTURES)),
          3 NAME                CHAR(LENGTH_KEY_NAME),
          3 TYPE                CHAR(1),
                                                       /* F = FIELD
                                                          G = GROUP */
          3 #SUBSCRIPTS         BIN FIXED(15),
          3 SUBSCRIPT1          BIN FIXED(15),
          3 SUBSCRIPT2          BIN FIXED(15),


          3 EXIST_PROC          CHAR(LENGTH_KEY_NAME),
          3 ARITY               BIN FIXED(15),
          3 DATA_TYPE           CHAR(1),
                                                       /* C = CHARACTER
                                                          B = BINARY
                                                          F = FIXED DECIMAL */
          3 FIELD_LEN_TYPE      CHAR(1),
                                                       /* F = FIXED
                                                          V = VARIABLE */
          3 MIN_LENGTH          BIN FIXED(15),
          3 MAX_LENGTH          BIN FIXED(15),
          3 LEN_PROC            CHAR(LENGTH_KEY_NAME);
```

```
-DCL CHPTRLB ENTRY(CHAR(*)) RETURNS(CHAR(LEN_DICT_ENTRY) VARYING);
 DCL SYSERR ENTRY(CHAR(*));
 DCL WRPL1 ENTRY(CHAR(*) VARYING, CHAR(*));
 DCL PICTURE ENTRY(BIN FIXED(15)) RETURNS(CHAR(10)VARYING);
 DCL SUBSCRIPT_STRING ENTRY RETURNS(CHAR(100) VARYING);
 DCL INDXGEN ENTRY(BIN FIXED(15)) RETURNS(CHAR(3));
 DCL PACK ENTRY(BIN FIXED(15));
 DCL UNPACK ENTRY(BIN FIXED(15));
 DCL PLISTR CHAR(320) VARYING;
     /* THIS IS THE VARIABLE IN WHICH WE FORM THE CODE FOR THE        */
     /* TARGET PROGRAM.                                               */
-/* THROUGHOUT THE COMMENTS TO THIS ROUTINE, CERTAIN META-VARIABLES   */
 /* WILL BE USED TO DESCRIBE THE CODE GENERATED.  THESE META-        */
 /* VARIABLES, DESCRIBED BELOW, ALWAYS BEGIN WITH THE CHARACTER      */
 /* '#'.                                                             */
0DCL #FILESUFF CHAR(MAX_L_NAME) VARYING;
     /* THIS IS FTR.FILENAME WITH TRAILING BLANKS DROPPED.  THE       */
     /* SUFFIX OF 'S' OR 'T' OR 'U' INDICATES THAT THIS IS A SOURCE   */
     /* E.G.  'SALEDECKS'  'INEVNU'  'MASTERS'  'MASTERT'.            */
0DCL #FILENAME CHAR(MAX_L_NAME) VARYING;
     /* THIS IS THE FILENAME WITHOUT THE SUFFIX.                      */
     /* E.G.  'SALEDECK'  'INVEN'  'MASTER'.                          */
0DCL #RECNAME CHAR(LEN_DICT_ENTRY) VARYING EXT;
     /* THIS IS THE RECORD NAME FROM FTR.RECNAME WITH TRAILING        */
     /* BLANKS DROPPED.  POSSIBLY PREFIXED WITH 'OLD.' OR 'NEW.'.     */
     /*  E.G.  'SALEREC'  'OLD.INVEN'  'NEW.INVEN'.                   */
0DCL #RECBAR CHAR(LEN_DICT_ENTRY) VARYING;
     /* THIS IS LIKE #RECNAME BUT HAS 'OLD_' AND 'NEW_' INSTEAD       */
     /* OF 'OLD.' AND 'NEW.'.  THIS IS USED FOR LABELS IN THE         */
     /* TARGET PROGRAM.                                               */
     /* E.G.  'SALEREC'  'OLD_INVEN'  'NEW_INVEN'                     */
0DCL #RECSTEM CHAR(LEN_DICT_ENTRY) VARYING;
     /* THIS IS THE "STEM" OF THE RECORD NAME, WITHOUT ANY PREFIX.    */
     /* USED IN REFERRING TO POINTERS IN THE TARGET PROGRAM.          */
     /* E.G.  'SALEREC'  'INVEN'.                                     */
 %DCL LEN_RECSTRING FIXED;
 %LEN_RECSTRING=2+LEN_DICT_ENTRY;
0DCL #RECSTRING CHAR(LEN_RECSTRING) VARYING EXT;
     /* THIS NAMES THE STRING VARIABLE *INTO* WHICH WE READ           */
     /* AND *FROM* WHICH WE WRITE IN THE TARGET PROGRAM.              */
     /* FORMED BY SUFFIXING #RECBAR WITH '_S'.                        */
     /* E.G.  'SALEREC_S'  'OLD_INVEN_S'  'NEW_INVEN_S'.              */
0DCL #KEYNAME CHAR(MAX_L_NAME) VARYING;
     /* THIS IS THE NAME OF THE KEY FIELD OF A RECORD.  USED FOR      */
     /* READING KEYED SEQUENTIAL FILES.  OBTAINED BY REMOVING         */
     /* TRAILING BLANKS FROM FLOWTAB_REC.KEYNAME.                     */
-DCL SUBSCRIPT_STACK(MAX_DEPTH_STACK) CHAR(3) EXT;
 DCL SUBSCRIPT_STACK_TOP BIN FIXED(15) EXT;
 DCL NSTR BIN FIXED(15) EXT;
 DCL FTR_PTR POINTER EXT;
1    FTR_PTR=FLOWTAB_REC_PTR;
```

```
     #FILESUFF=CHPTRLB(FTR.FILENAME);
     #FILENAME=SUBSTR(#FILESUFF,1,LENGTH(#FILESUFF)-1);
0/* #RECNAME COMES DIRECTLY FROM FTR, WITH TRAILING BLANKS DROPPED.     */
     #RECNAME=CHPTRLB(FTR.RECNAME);
0/* GET THE STEM OF #RECNAME.                                            */
     IF 1=INDEX(#RECNAME,'OLD.')
     THEN #RECSTEM=SUBSTR(#RECNAME,5,LENGTH(#RECNAME)-4);
     ELSE IF 1=INDEX(#RECNAME,'NEW.')
          THEN #RECSTEM=SUBSTR(#RECNAME,5,LENGTH(#RECNAME)-4);
          ELSE /* NO PREFIX */
               #RECSTEM=#RECNAME;
0/* FORM #RECBAR.                                                       */
 /* E.G.       'SALEREC'       -->      'SALEREC'                       */
 /*            'NEW.INVEN'     -->      'NEW_INVEN'                     */
 /*            'OLD.INVEN'     -->      'OLD_INVEN'                     */
     IF 1=INDEX(#RECNAME,'OLD.')
     THEN #RECBAR='OLD_' || #RECSTEM;
     ELSE IF 1=INDEX(#RECNAME,'NEW.')
          THEN #RECBAR='NEW_' || #RECSTEM;
          ELSE /* NO PREFIX */
               #RECBAR=#RECSTEM;
0/* FORM #RECSTRING.                                                    */
     #RECSTRING=#RECBAR || '_S';
0/* FORM #KEYNAME.                                                      */
     #KEYNAME=CHPTRLB(FTR.KEYNAME);
 PUT SKIP LIST('*********************');                                 DEBUG
 PUT SKIP LIST('* GENIOCD CALLED *');                                    DEBUG
 PUT SKIP LIST('*********************');                                 DEBUG
 PLISTR='DO;';
 CALL WRPLI(PLISTR,'PLIEX');
-/* BRANCH TO LABEL WHICH HANDLES THIS PARTICULAR TYPE OF FLOWTAB_REC.*/
0    IF (IOMODE='RD') & (ORG='S') & (KEYED=0) THEN GOTO CASE1;
0    IF (IOMODE='RD') & (ORG='S') & (KEYED=1) THEN GOTO CASE2;
0    IF (IOMODE='RD') & (ORG='I') & (KEYED=1) THEN GOTO CASE3;
0    IF (IOMODE='WR') & (ORG='S') THEN GOTO CASE4;
0    IF (IOMODE='RW') & (ORG='I') & (KEYED=1) THEN GOTO CASE5;
0/* IF NO CASE APPLIES, THEN SYSTEM ERROR                               */
     CALL SYSERR('GENIOCD:  UNKNOWN FORM OF FLOWTAB_REC');
     RETURN; /* NOT NEEDED, AS SYSERR WILL HALT EXECUTION */
1CASE1:
-/* THIS SECTION WRITES CODE FOR         IOMODE='RD'                    */
 /*                                      ORG='S'                        */
 /*                                      UNKEYED                        */
-/* FILE PLIEX GETS:                                                    */
0/*    $RD_"#FILESUFF":                                                 */
 /*    READ FILE("#FILESUFF") INTO("#RECSTRING");                       */
-/* FILE PLION GETS:                                                    */
0/*    ON ENDFILE("#FILESUFF") GOTO $FINISH;                            */
-PLISTR='$RD_' || #FILESUFF || ':';
 CALL WRPLI(PLISTR,'PLIEX');
0PLISTR='READ FILE(' || #FILESUFF || ') INTO (' || #RECSTRING || ');';
 CALL WRPLI(PLISTR,'PLIEX');
0IF (FTR.PACKED=1) THEN CALL GENERATE_UNPACKING;
0PLISTR='ON ENDFILE (' || #FILESUFF || ') GOTO $FINISH;';
 CALL WRPLI(PLISTR,'PLION');
0GOTO FINISHED;
1CASE2:
-/* THIS SECTION WRITES CODE FOR         IOMODE='RD'                    */
 /*                                      ORG='S'                        */
 /*                                      KEYED                          */
-/* FILE PLIEX GETS:                                                    */
0/*    $RD_"#FILESUFF":                                                 */
 /*    READ FILE("#FILESUFF")INTO("#RECSTRING");                        */
 /*    IF "#FILENAME"."#KEYNAME" = POINTER."#RECSTEM"                    */
 /*    THEN GOTO $"#RFCBAR"_END;                                         */
 /*    IF "#FILENAME"."#KEYNAME" < POINTER."#RECSTEM"                    */
 /*    THEN DO;                                                          */
 /*          WRITE FILE("#FILENAME"T) FPCM("#RECSTRING");                */
 /*          GOTO $RD_"#FILESUFF";                                       */
 /*          END;                                                        */
 /*    CALL $NCTFOUND('"#FILESUFF"');                                    */
 /*    $"#RECBAR"_END:                                                   */
-/* FILE PLION GETS:                                                     */
 /*    ON ENDFILE("#FILESUFF") CALL $NCTFCUND('"#FILESUFF"');            */
-PLISTR='SRD_' || #FILESUFF || ':';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='READ FILE(' || #FILESUFF || ') INTC (' || #RECSTRING || ');';
 CALL WRPLI(PLISTR,'PLIEX');
OIF (FTR.PACKED=1) THEN CALL GENERATE_UNPACKING;
OPLISTR='IF ' || #FILENAME || '.' || #KEYNAME || ' = POINTER.' ||
        #RECSTEM || ' THEN GOTO $' || #RECBAR || '_END;';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='IF ' || #FILENAME || '.' ||#KEYNAME || ' < PCINTER.' ||
        #RECSTEM || ' THEN DO;';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='WRITE FILE(' || #FILENAME || 'T) FRCM(' || #RECSTRING || ');';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='GOTO $RD_' || #FILESUFF || ';';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='END;';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='CALL $NCTFOUND(''' || #FILESUFF || ''');';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='$' || #RECBAR || '_END:';
 CALL WRPLI(PLISTR,'PLIEX');
OPLISTR='ON ENDFILE(' || #FILESUFF ||
               ') CALL $NOTFCUND(''' ||
               #FILESUFF || ''');';
 CALL WRPLI(PLISTR,'PLION');
OGCTC FINISHED;
1CASE3:
-/* THIS SECTION WRITES CCDE FCR        ICMCDE='RD'                      */
 /*                                    CRG='I'                          */
 /*                                    KEYED          (NECESSARILY)     */
-/* FILE PLIEX GETS:                                                     */
0/*    READ FILE("#FILESUFF") INTC("#RECSTRING")                         */
 /*          KEY(POINTER."#RECSTEM");                                    */
-/* FILE PLION GETS:                                                     */
0/*    ON KEY("#FILESUFF") CALL $NCTFCUND('"#FILESUFF"');                */
-PLISTR='READ FILE(' || #FILESUFF || ') INTC (' ||
               #RECSTRING || ') KEY(PCINTER.' ||
               #RECSTEM || ');';
 CALL WRPLI(PLISTR,'PLIEX');
OIF (FTR.PACKED=1) THEN CALL GENERATE_UNPACKING;
OPLISTR='ON KEY(' || #FILESUFF ||
               ') CALL $NOTFOUNC(''' || #FILESUFF || ''');';
 CALL WRPLI(PLISTR,'PLION');
CGCTC FINISHED;
1CASE4:
-/* THIS SECTION WRITES CCDE FCR        ICMCDE='WR'                      */
 /*                                    CRG='S'                          */
 /*                                    KEYED OR UNKEYED                 */
-/* FILE PLIEX GETS:                                                     */
0/*    WRITE FILE("#FILESUFF") FROM("#RECSTRING");                       */
-IF (FTR.PACKED=1) THEN CALL GENERATE_PACKING;
OPLISTR='WRITE FILE(' || #FILESUFF || ') FRCM (' || #RECSTRING || ');';
```

```
 CALL WRPLI(PLISTR,'PLIEX');
0GOTO FINISHED;
1CASE5:
-/* THIS SECTION WRITES CODE FOR          IOMODE='RW'                    */
 /*                                      ORG='I'                        */
 /*                                      KEYED      (NECESSARILY)       */
-/* FILE PLIEX GETS:                                                     */
0/*    REWRITE FILE("#FILESUFF") FROM("#RECSTRING")                     */
 /*             KEY(POINTER."#RECSTEM");                                */
-IF (FTR.PACKED=1) THEN CALL GENERATE_PACKING;
0PLISTR='REWRITE FILE(' || #FILESUFF ||
             ') FROM (' || #RECSTRING ||
             ') KEY(POINTER.' || #RECSTEM || ');';
 CALL WRPLI(PLISTR,'PLIEX');
-FINISHED:
 PLISTR='END;';
 CALL WRPLI(PLISTR,'PLIEX');
 PUT EDIT('***** EXIT GENIOCD *****')(SKIP,A);
 RETURN;
1GENERATE_UNPACKING:  PROC;
 /* THIS IS THE SUPERVISORY PROCEDURE FOR GENERATING UNPACKING          */
 /* INSTRUCTIONS.                                                       */
 DCL I                  BIN FIXED(15);
     CALL INITIALIZE_SUBSCRIPT_STACK;                                          1
     NSTR=0; /* TELLS WHICH SUBSTRUCTURE WE'RE LOOKING AT.              */     2
     PLISTR='I=1;'; /* INITIALIZE BUFFER POINTER. */                           3
     CALL WRPLI(PLISTR,'PLIEX');
     DO I=1 TO RECORD_ARITY;
          CALL UNPACK(1); /* NEXT_INDEX TO USE IS 1. */                        4
     END;
 RETURN;
 END; /* GENERATE_UNPACKING */
1GENERATE_PACKING:  PROC;
 /* THIS IS THE SUPERVISORY PROCEDURE FOR GENERATING CODE TO PACK       */
 /* OUTPUT RECORDS.                                                     */
 DCL I                  BIN FIXED(15);
     CALL INITIALIZE_SUBSCRIPT_STACK;                                              1
     NSTR=0;                                                                       2
 /* GENERATE CODE TO INITIALIZE OUTPUT STRING.                          */
     PLISTR=#RECSTRING ||
        '='''';';                                                                  3
     CALL WRPLI(PLISTR,'PLIEX');
 /* CALL PACK FOR EACH MEMBER OF THE RECORD.                            */
     DO I=1 TO RECORD_ARITY;
          CALL PACK(1); /* NEXT_INDEX TO USE IS 1. */                              4
     END;
 RETURN;
 END; /* GENERATE_PACKING */
-INITIALIZE_SUBSCRIPT_STACK:  PROC;
 /* INTERNAL TO GENIOCD.                                                */
 /* THIS PROCEDURE INITIALIZES THE SUBSCRIPT_STACK;  CALLED BY:         */
 /*              GENERATE_PACKING                                       */
 /*              GENERATE_UNPACKING.                                    */
     SUBSCRIPT_STACK_TOP=0;
 RETURN;
 END; /* INITIALIZE_SUBSCRIPT_STACK */
 END; /* GENIOCD */
```

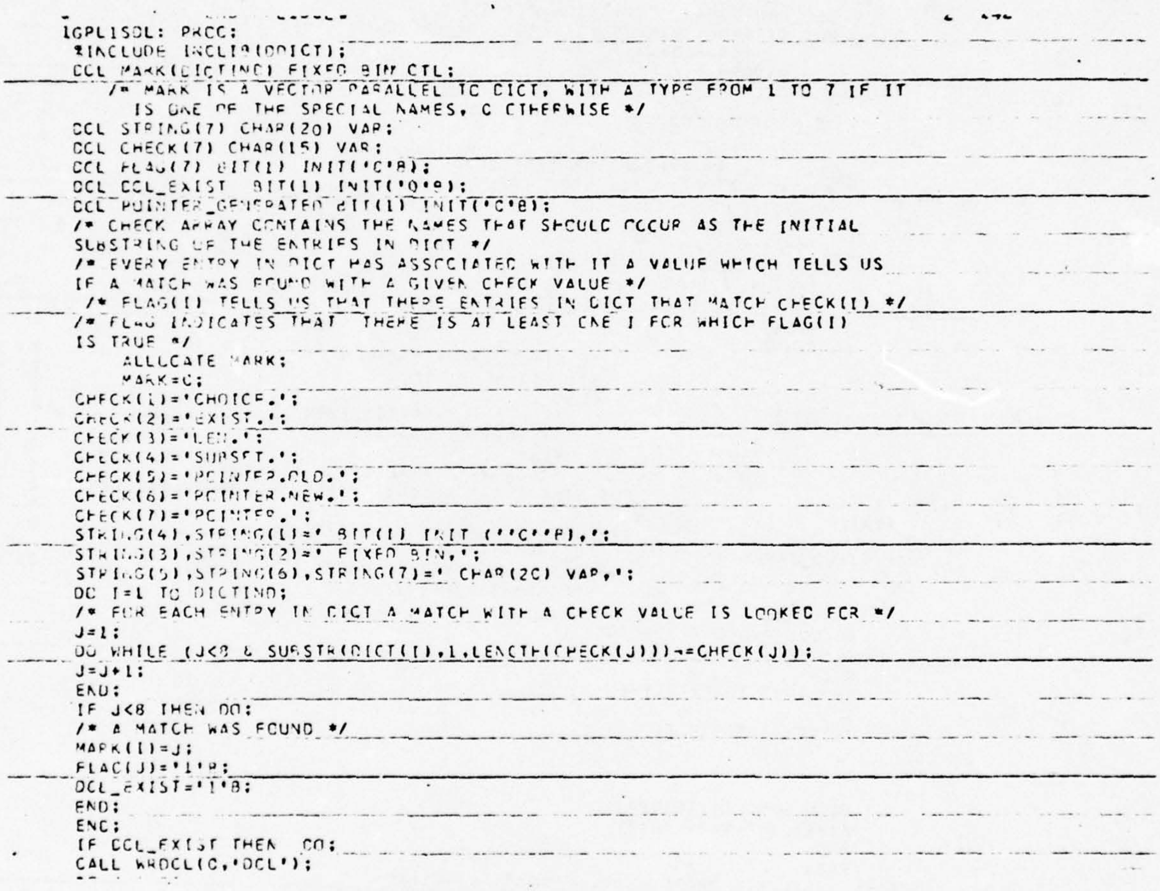
```
*PROCESS('NST,MACRO,N=GENRSET');
 GENRSET: PROC(FLOW_PTR);
      %DCL MAX_LEN_QNAME FIXED;
      %MAX_LEN_QNAME=32;
 /*THIS PROCEDURE RESETS SWITCHES AT END OF EACH READ CYCLE */
      DCL FLOW_PTR POINTER;
      DCL TEMP_ZERO CHAR(4) VAR;
      DCL CHPTPLB ENTRY(CHAR(*)) RETURNS(CHAR(MAX_LEN_QNAME)VAR);
      DCL 1 FLOWTAB_RSET BASED(P),
           2 NODE# FIXED BIN,
           2 TYPE CHAR(4),
           2 NAME CHAR(MAX_LEN_QNAME);
      P=FLOW_PTR;
      IF NAME='#STACK' THEN /* THIS IS ONLY RUNTIME VARIABLE THAT IS
      FIXED BIN RATHER THAN BIT(1); IT IS INDEX  TO RUNTIME STACK FOR
      REPLACEMENT  */
       TEMP_ZERO='0';  ELSE TEMP_ZERO='''0''B';
      CALL WRPL1(CHPTPLB(NAME)||'='||TEMP_ZERO||';','PL1EX');
 END;

*PROCESS('NST,N=GENTEXT');
 GENTEXT: PROC(P);
 DCL 1 FLOWTAB_TEXT BASED(FLOWTAB_TEXT_PTR),
       2 NODE# FIXED BIN,
       2 TYPE CHAR(4), /* 'TEXT' */
       2 LEN_TEXT FIXED BIN,
       2 PL1_STR CHAR(N REFER(LEN_TEXT));


 DCL P POINTER;
      FLOWTAB_TEXT_PTR=P;
   CALL WRPL1(PL1_STR,'PL1EX');
 END;
```

```
*PROCESS('NST,MACRO,N=GIMFLD');
 GIMFLD:PROC(TAB_FLD_PTR);
 /* GENERATE ASSIGNMENT STATEMENTS FOR IMPLICIT FIELD ASSOCIATIONS. */
 DECLARE TAB_FLD_PTR POINTER;
 %INCLUDE INCLIB(DDICT);
 DCL CHPTRLB ENTRY(CHAR(*)) RETURNS(CHAR(LEN_DICT_ENTRY)VAR);
 DCL 1 FLOWTAB_FIELD BASED(FLOWTAB_FIELD_PTR),
          2 NODE# FIXED BIN,
          2 NODE_TYPE CHAR(4),
          2 TGT_FIELD CHAR(LEN_DICT_ENTRY),
          2 SRCE_FIELD CHAR(LEN_DICT_ENTRY);
 FLOWTAB_FIELD_PTR= TAB_FLD_PTR;
 IF NODE_TYPE¬='FLDI'& NODE_TYPE ¬= 'INTI' THEN
      CALL SYSERR('GIMFLD: BAD TYPE CODE: '||NODE_TYPE);
    ELSE CALL WRPLI(CHPTRLB(TGT_FIELD)||'='||CHPTRLB(SRCE_FIELD)||';',
          'PLIEX');
 RETURN;
 END GIMFLD;

*PROCESS('NST,MACRO,N=GMODCD');
 GMODCD:PROC(PSSD_PTR);
 /* THIS PROCEDURE GENERATES THE PROCEDURE STATEMENT FOR A MODULE. */
 DCL PSSD_PTR POINTER;
 DCL PROC_STMT  CHAR(80);
 DCL 1 FLOWTAB_MOD BASED (FLOW_PTR),
          2 NODE# FIXED BIN,
          2 NODE_TYPE CHAR(4),
          2 MODULE_NAME CHAR(10);
 FLOW_PTR=PSSD_PTR;
 IF NODE_TYPE¬='MODL' THEN CALL SYSERR('GMODCD: BAD CODE:'||NODE_TYPE);
    ELSE
   DO;
      PROC_STMT=' '||MODULE_NAME||' : PROC OPTIONS(MAIN);';
      WRITE FILE(PLIDCL)  FROM(PROC_STMT);
   END;
 RETURN;
 END GMODCD;
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=GPLICCL');
 GPLIDCL: PROC;
 /* PLICCL GENERATES A PLI DECLARATION FOR EVERY DCL_PROTO DECLARATION     1   2
 */        %DCL LENGTH_KEY_NAME FIXED;                                     1   2
           %DCL MAX_LEN_QNM   FIXED;
 %DCL MAX#_CONDAS FIXED;
 %DCL MAX_LEN_FOREACH FIXED;
           %MAX_LEN_QNM  =32;
 %MAX#_CONDAS=30;
 %MAX_LEN_FOREACH=18;
           %LENGTH_KEY_NAME=10;                                            1   3
           DCL LINE CHAR(284) VAR;
 /* LINE IS EQUIVALENT TO A PUNCHED CARD . IT REPRESENTS A PLI DCL */      1   5
           DCL SIGNAL BIT(1);                                              1   5
 /* SIGNAL='0'B INDICATES THAT THERE ARE NC UPDATE FILES */                1   6
 /* ON THIS FILE THE LINE IS WRITTEN */                                    1   7
           DCL NULL BUILTIN;                                               1   7
0          DCL 1 FIELD_DCL_PROTO BASED (P),
           /* THIS IS THE PROTOTYPE DECLARATION OF A FIELD OR INTERIM */
           2 TYPE CHAR(2),     /* 'FD' FOR FIELD OR 'IN' FOR INTERIM */


           2 FIELD_LEVEL FIXED (3) DEC,
           2 FIELD_NAME CHAR(LENGTH_KEY_NAME),                             1  11
           2 MAX_REPETITION FIXED DEC(3),
           2 FIELD_TYPE CHAR(1),
           2 FIELD_LEN_TYPE CHAR(1),
           2 MAX_LEN FIXED (5,0) DECIMAL,
           2 MIN_LEN FIXED (5,0) DECIMAL  ;
0          DCL 1 FILE_DCL_PROTO BASED (P),
            /* THIS IS THE PROTOTYPE DECLARATION OF A FILE OR A REPORT */  1  12
           2 TYPE CHAR(2),     /* 'FL' FOR FILE DECLARATION */
           2 FILE_LEVEL FIXED (3) DEC,
           2 FILE_NAME CHAR (LENGTH_KEY_NAME),                             1  12
           2 FILE_FLOW CHAR(1),                                            1  12
           2 FILE_ORG CHAR(1);                                             1  12
0          DCL 1 RECORD_DCL_PROTO BASED (P),                               1  13
           /* THIS IS THE PROTOTYPE  DECLARATION OF  A RECORD OR A REPORT  1  13
           ENTRY */                                                        1  13
           2 TYPE CHAR(2),     /* 'RC' FOR RECORD STRING DECLARATION  */
           2 RECORD_LEVEL FIXED (3,C) DECIMAL,
           2 RECORD_NAME CHAR(MAX_LEN_QNM),
           2 RECORD_LENGTH_TYPE CHAR(1),                                   1  13
           2 RECORD_LENGTH FIXED DEC(5,C),
           2 DEF_RECNAME CHAR(MAX_LEN_QNM);
0          DCL 1 DCL_PROTO BASED (P),                                      1  14
           /* THIS GENERAL DECLARATION IS USED FOR STRUCTURES */           1  14
           2 TYPE CHAR(2),/* 'FR' FOR FILE NAME AT TOP LEVEL OF STRUCTURE;
                             'RR' FOR RECORD DCL AT 2ND LEVEL OF STRUCTURE
                             'GR' FOR GROUP DCL IN THE STRUCTURE  */
           2 LEVEL FIXED (3,0) DEC,
           2 NAME CHAR(LENGTH_KEY_NAME),                                   1  14
           2 MAX_REPETITION FIXED DEC(3);
      DCL CONDAS(MAX#_CONDAS) CHAR(LENGTH_KEY_NAME) VAR EXT;
        /*TABLE OF ASSERTIONS USING CONDITIONAL FUNCTIONS*/
      DCL #CONDAS FIXED BIN EXT;
  %INCLUDE INCLIB(DSYSFCN);
-          DCL CHPTPLD ENTRY(CHAR(*)) RETURNS(CHAR(MAX_LEN_QNM)VAR);
0          DCL ENDFILE_DCLTAB BIT(1)INIT('0'B);
 DCL FIRST_INTERIM BIT(1) INIT('1'B);
 DCL 1 DOTAB(*)CTL EXT,
      2 LOC FIXED BIN,
      2 DO_LOOP_VAR CHAR(MAX_LEN_FOREACH);
 DCL #LOOPS FIXED BIN EXT;
           DCL WRDCL ENTRY(FIXED DEC,CHAR(*)VAR);
 /* DCLTAB IS THE INPUT FILE ON WHICH THE PRCTO_CCL ARE STOREO */         1   14
OCPEN FILE(DCLTAR) INPUT RECCPC SEQL;
0        ON ENDFILE(DCLTAR) ENDFILE_DCLTAP='1'B;
-/* INITIALLY THERE ARE NO UPDATE FILES */                                1   15
          SIGNAL='0'B;
-/* GENERATE DCL KEYWORD: */
            LINE=' DCL  ';                                                1   15
         CALL WRDCL(0,LINE);
0        READ FILE(DCLTAR) SET(P);
0           DO WHILE(¬ENDFILE_DCLTAB);
             CALL FPLIDCL;                                                2   19
             READ FILE(DCLTAB)SET(P);                                     2   18
 /* READ THE PROTO DCL AND CALL FPLIDCL UNTIL THE END OF DCLTAB */        2   20
             END;                                                         2   20
0            IF SIGNAL='1'B THEN                                          2   21
 DO;
 /*SIGNAL FOR PRESENCE OF I/O FILES*/
 /* DECLARE NEW LIKE OLD */                                               3   25
 LINE='NEW LIKE OLD;';
 CALL WRDCL(1,LINE);


 END;
 ELSE CALL WRDCL(0,'I FIXED BIN;');
-      IF USEFCN(1) THEN  /* USED THE 'REPLACE' FCN; GENERATE DCLS */
         DO;
           CALL WRDCL(0,'DCL');
           CALL WRDCL(0,'WASREPL BIT(1) INIT(''C''B),');
           CALL WRDCL(0,'AIRREPL BIT(1)INIT(''C''B),');
           CALL WRDCL(0,'$STACK(5C)CHAR(1C)VAR,');
           CALL WRDCL(0,'#STACK FIXED BIN INIT(0);');
         END;
-/* DCL 'SELECTED' AND 'NOT_SELECTED' RUN-TIME VARIABLES */
      CALL WRDCL(0,'DCL SELECTEC BIT(1) INIT(''1''B);');
      CALL WRDCL(0,'DCL NOT_SELECTEC BIT(1) INIT(''C''B);');
-/*GENERATE DCL FOR ANY CONDITIONAL FCN USED */
ODO I=1 TO #SYSFCN;
      IF CONDF(I) & USEFCN(I) THEN
         CALL WRDCL(0,'DCL '||CHPTFLB(SYSFCN(I))||'_COMPLETED BIT(1);');
  END;
-/*GENERATE DCL IF ALL CONDITIONAL-ASSERTIONS COMPLETION FLAG*/
      DO I=1 TO #CONDAS;
  CALL WRDCL(0,'DCL '||CONDAS(I)||'_COMPLETED BIT(1)INIT(''C''B);');
      END;
-/*GENERATE DCL FOR "DO" ("FOREACH") VARIABLES */
0      DO I=1 TO #LOOPS;
         CALL WRDCL(0,'DCL '||DC_LOOP_VAR(I)||' FIXED BIN;');
      END;
```

```
-/* GENERATE DECLARATIONS OF 'SPECIAL' NAMES: I.E. 'CHOICE', 'EXIST',
     'LEN', 'POINTER', AND 'SUBSET' DECLARATIONS */
0     CALL GPLISDL;
-CLOSE FILE(PL1DCL);
1FPL1DCL:    PROC;                                                          2   26
0            DCL                                                            2   27
             (STRING1,STRING2) CHAR(32) VAR STATIC,
            /* STRING1 AND STRING2 ARE USED AS TEMPORARY SUBSTRINGS OF
            OF LINE*/
             STRING CHAR(5)  STATIC;
             /* STRING IS A SUBSTRING OF LINE WHICH WILL BE THE STRING EX  2   27
 PRESSION    OF A DECIMAL NUMBER */                                        2   27
 DCL NINES CHAR(15) INIT((15)'9');
 DCL ILINE CHAR(13) VAR STATIC;
 /* TEMPORARY STRING FOR 'INTERIM' DCL  */
0                 IF DCL_PROTO.TYPE='FL' THEN                              3   28
                     DO;                                                   4   29
                        IF FILE_DCL_PROTO.FILE_FLOW='I' THEN               5   30
                        STRING1=' INPUT ';                                 5   31
                        ELSE                                               5   32
                           IF FILE_DCL_PROTO.FILE_FLOW='O' THEN            6   32
                           STRING1=' OUTPUT ';                             6   33
                           ELSE                                            6   34
                           STRING1=' UPDATE ';                             6   34
                        IF FILE_DCL_PROTO.FILE_ORG='S' THEN                5   35
                        STRING2=' RECORD SEQL ';
                        ELSE                                               5   37
                  STRING2=' KEYED DIRECT ENV(INDEXED) ';
 LINE=FILE_DCL_PROTO.FILE_NAME||' FILE'||STRING1||STRING2||',';
 CALL WRDCL (O,LINE);
                     END;                                                  4   41
0                 ELSE
                  IF DCL_PROTO.TYPE='RC' THEN                              3   42
                     DO;                                                   4   43
                        IF RECORD_DCL_PROTO.RECORD_LENGTH_TYPE='V' THEN    5   45
                        STRING2='VAR';                                     5   46
                        ELSE                                               5   47


                        STRING2='DEF '||CHPTRLB(DEF_RECNAME);
                     PUT STRING(STRING)EDIT(RECORD_DCL_PROTO.RECORD_LENGTH) 4   48
 (F(5));                                                                   4   48
 LINE=CHPTRLB(RECORD_DCL_PROTO.RECORD_NAME)||' CHAR('||STRING||') '
 ||STRING2||',';
   CALL WRDCL (O,LINE);
                     END;                                                  4   60
0                 ELSE
                  IF SIGNAL='0'B&DCL_PROTO.TYPE='FR'&DCL_PROTO.LEVEL=2 THEN 3  61
                  /*FIRST STRUCTURE FOR AN I/O FILE*/
                                                                           3   61
```

```
                                                                              3   61
                    DO;                                                       4   62
      /* THERE ARE UPDATE FILES */                                            4   63
                    SIGNAL='1'B;                                              4   63
                    LINE='DCL,';
                    CALL WRDCL(1,LINE);
                    LINE=DCL_PROTO.NAME||',';
                    CALL WRDCL(2,LINE);
                    END;                                                      4   68
 -              ELSE                                                          3   69
                      /* ALL OTHER TYPES  */
                    DO;                                                       4   69
 0                    IF DCL_PROTO.TYPE¬='FD' & DCL_PROTO.TYPE¬='IN' THEN
                  /*PROCESS ALL OTHER TYPES BUT FIELDS & INTERIMS */
      DO;
      IF DCL_PROTO.MAX_REPETITION>0 THEN
         PUT STRING(STRING2) EDIT('(',DCL_PROTO.MAX_REPETITION,')')
               (A,F(3),A);
      ELSE STRING2='';
      CALL WRDCL(DCL_PROTO.LEVEL,DCL_PROTO.NAME|| STRING2||',');
   END;
 0                      ELSE                                                  5   93
                           DO;                                                6   93
                  /* TYPE=FD OR TYPE=IN*/
      IF FIELD_DCL_PROTO.MAX_REPETITION>0  THEN
         PUT STRING(STRING2) EDIT('(',FIELD_DCL_PROTO.MAX_REPETITION,')')
               (A,F(3),A);
      ELSE STRING2='';
    LINE=DCL_PROTO.NAME||STRING2;
          IF FIELD_TYPE ='N' THEN STRING1=' PIC'''||
          SUBSTR(NINES,1,FIELD_DCL_PROTO.MAX_LEN)||''',';
          ELSE
          DO;
                                IF FIELD_TYPE='B' THEN                        7  105
                                STRING1='  BIN(';                             7  106
                                ELSE                                          7  107
                                  IF FIELD_TYPE='C' THEN                      8  107
                                  STRING1='  CHAR(';                          8  108
                                  ELSE                                        8  109
                                  STRING1='  FIXED(';                         8  109
                             PUT STRING(STRING)EDIT(FIELD_DCL_PROTO.MAX_LEN)( 6  110
   F(5));                                                                     6  110
                             STRING1=STRING1||STRING||')';                    6  111
   IF FIELD_TYPE='C' &FIELD_LEN_TYPE='V' THEN
                                STRING1=STRING1||' VAR';                      7  113
                                STRING1=STRING1||',';
          END;
      IF DCL_PROTO.TYPE='IN'
      THEN
      DO;
      IF FIRST_INTERIM THEN
      DO;
      ILINE='INTERIM,';


      CALL WRDCL(1,ILINE);
      FIRST_INTERIM= '0'B;
      END;
      END;
   CALL WRDCL(DCL_PROTO.LEVEL,LINE||STRING1);
                             END;                                             6  124
                       END;                                                   4  125
 0               END FRLIDCL;                                                 2  142
```



```
IGPLISDL: PROC;
 %INCLUDE INCLIB(DDICT);
 DCL MARK(DICTIND) FIXED BIN CTL;
    /* MARK IS A VECTOR PARALLEL TO DICT, WITH A TYPE FROM 1 TO 7 IF IT
      IS ONE OF THE SPECIAL NAMES, 0 OTHERWISE */
 DCL STRING(7) CHAR(20) VAR;
 DCL CHECK(7) CHAR(15) VAR;
 DCL FLAG(7) BIT(1) INIT('0'B);
 DCL DCL_EXIST  BIT(1) INIT('0'B);
 DCL POINTER_GENERATED BIT(1) INIT('0'B);
 /* CHECK ARRAY CONTAINS THE NAMES THAT SHOULD OCCUR AS THE INITIAL
 SUBSTRING OF THE ENTRIES IN DICT */
 /* EVERY ENTRY IN DICT HAS ASSOCIATED WITH IT A VALUE WHICH TELLS US
 IF A MATCH WAS FOUND WITH A GIVEN CHECK VALUE */
  /* FLAG(I) TELLS US THAT THERE ENTRIES IN DICT THAT MATCH CHECK(I) */
 /* FLAG INDICATES THAT  THERE IS AT LEAST ONE I FOR WHICH FLAG(I)
 IS TRUE */
     ALLOCATE MARK;
     MARK=0;
 CHECK(1)='CHOICE.';
 CHECK(2)='EXIST.';
 CHECK(3)='LEN.';
 CHECK(4)='SUBSET.';
 CHECK(5)='POINTER.OLD.';
 CHECK(6)='POINTER.NEW.';
 CHECK(7)='POINTER.';
 STRING(4),STRING(1)=' BIT(1) INIT (''0''B),';
 STRING(3),STRING(2)=' FIXED BIN,';
 STRING(5),STRING(6),STRING(7)=' CHAR(20) VAR,';
 DO I=1 TO DICTIND;
 /* FOR EACH ENTRY IN DICT A MATCH WITH A CHECK VALUE IS LOOKED FOR */
 J=1;
 DO WHILE (J<8 & SUBSTR(DICT(I),1,LENGTH(CHECK(J)))¬=CHECK(J));
 J=J+1;
 END;
 IF J<8 THEN DO;
 /* A MATCH WAS FOUND */
 MARK(I)=J;
 FLAG(J)='1'B;
 DCL_EXIST='1'B;
 END;
 END;
 IF DCL_EXIST THEN  DO;
 CALL WRDCL(0,'DCL');
```

```
DO J=1 TO 7;
IF FLAG(J) THEN DO;
IF J<5 THEN
 DO;
LINE=SUBSTR(CHECK(J),1,LENGTH(CHECK(J))-1)||',';
 CALL WRDCL(1,LINE);
 END;
ELSE IF ¬ POINTER_GENERATED THEN DO;
   POINTER_GENERATED='1'B;
LINE='POINTER,';
CALL WRDCL(1,LINE);


  END;
IF J=5 THEN DO;
LINE='OLD,';
CALL WRDCL(2,LINE);
END;
IF J=6 THEN DO;
LINE='NEW,';
CALL WRDCL(2,LINE);
END;
IF J=5 | J=6 THEN K=3;
ELSE K=2;
/* K REPRESENTS THE LEVEL IN THE STRUCTURE OF THE NAME TAKEN FROM
DICT */
DO I=1 TO DICTIND;
LINE= SUBSTR(DICT(I),LENGTH(CHECK(J))+1)||STRING(J);
IF MARK(I)=J THEN CALL WRDCL(K,LINE);
END;
END;
END;
CALL WRDCL(0,'SY BIT(1);');
/* FOR ADDING THE SEMICOLON */
END;
END GPLISDL;
-END GPLIDCL;
```

```
*PROCESS('NST,N=GPROCCD,MACRO,SM=(2,72,1)');
 GPROCCD: PROC(PTR_TO_FLOWTAB);
 /* THIS PROCEDURE GENERATES 2 SETS OF CODE FOR ASSERTIONS:
      1) A CALL TO THE ASSERTION IN 'PL1EX'
 AND
 2) THE PROCEDURE ITSELF INVOLVING THE TEXT IN 'PL1PROC'. */
 %DCL MAX_LEN_QNM FIXED;
 %MAX_LEN_QNM=32;
 DCL PTR_TO_FLOWTAB PTR;
 %DCL(MAX_FN_NAME, MAX_L_NAME) FIXED;
 %MAX_FN_NAME=7;
 %MAX_L_NAME=10;
 /* THE RECORD WHICH IS POINTED TO BY THE ARGUMENT PASSED. */
 DCL      1 FLOWTAB_ASSN BASED (FLOW_PTR),
                  2 NODE# FIXED BIN,
                  2 NODE_TYPE CHAR(4),               /* ASSN */
                  2 ASSERTION_NAME CHAR(MAX_L_NAME),
                  2 COND_FCN CHAR(MAX_FN_NAME),
                  2 REPLACE_LABEL  CHAR(5),
                  2 ALREADY_CALLED BIT(1),
                  2 LEN_TEXT FIXED BIN,
                  2 TEXT CHAR(N REFER (LEN_TEXT));
 DCL SHORT_NAME CHAR(MAX_L_NAME) VAR;
 DCL PLPC CHAR(7) INIT('PL1PROC');
 DCL COND ENTRY(CHAR(*))RETURNS(BIT(1));
 DCL CHPTRLB ENTRY(CHAR(*)) RETURNS(CHAR(MAX_LEN_QNM)VAR);
 /* FETCH THE RECORD. */
 FLOW_PTR=PTR_TO_FLOWTAB;
 /* TO GENERATE THE CALL ALL WE NEED IS THE NAME. */
 SHORT_NAME=CHPTRLB(ASSERTION_NAME);
-IF  ¬ALREADY_CALLED THEN
0CALL WRPL1('CALL '||SHORT_NAME||';','PL1EX');
0/* PROCEDURE HAS NOT BEEN CALLED: CHECK IF INVOLVECTHE 'REPLACE'
    FUNCTION, AND IF SO, GENERATE THE APPROPRIATE CODE: */
 IF COND_FCN¬=' ' THEN
 DO;
 IF COND_FCN='REPLACE' THEN
 DO;


     CALL WRPL1('IF WASREPL THEN','PL1EX');
     CALL WRPL1('IF ¬CHOICE.EMPTY THEN DO;','PL1EX');
   CALL WRPL1('WASREPL=''0''B;','PL1EX');
     CALL WRPL1('GO TO '||REPLACE_LABEL||';','PL1EX');
   CALL WRPL1('END;','PL1EX');
 END;
 ELSE
 IF COND(COND_FCN) THEN
 DO;
  /*ASSERTION IS USING A CONDITIONALLY COMPLETED FCN*/
     CALL WRPL1('IF '||CHPTRLB(COND_FCN)||
     '_COMPLETED THEN DO;','PL1EX');
     CALL WRPL1(CHPTRLB(COND_FCN)||'_COMPLETED=''0''B;','PL1EX');
     CALL WRPL1(CHPTRLB(ASSERTION_NAME)||'_COMPLETED=''1''B;','PL1EX');
     CALL WRPL1('END;','PL1EX');
     END;
 END;
0/* TO GENERATE THE PROCEDURE WE NEED THE TEXT - IT IS ALREADY CLEANED
 UP. */
 CALL WRPL1(SHORT_NAME||': PROCEDURE;',PLPC);
 CALL WRPL1(TEXT,PLPC);
 CALL WRPL1('END '||SHORT_NAME||';',PLPC);
 END GPROCCD;
```

```
*PROCESS('NST,N=IDEND');
 IDEND: PROC;
      DCL 1 FLOWTAB_END BASED(FLOWTAB_END_PTR),
            2 NODE# FIXED BIN,
            2 TYPE CHAR(4);
      LOCATE FLOWTAB_END FILE(FLOWTAB);
      NODE#=0;
      TYPE='END';
 END IDEND;
*PROCESS('NST,MACRO,N=IDFIN');
 IDFIN: PROC;
 %DCL MAX_LEN_LABEL  FIXED;
 %MAX_LEN_LABEL=14;
      DCL 1 FLOWTAB_LAB  BASED(P),
            2 NODE# FIXED BIN,
            2 TYPE CHAR(4),
                 /* 'LAB' */
            2 LABEL CHAR(MAX_LEN_LABEL);
      LOCATE FLOWTAB_LAB  FILE(FLOWTAB);
            NODE#=0;
            TYPE='LAB';
            LABEL='$FINISH';
      END IDFIN;
```

```
*PROCESS('NST,MACRO,N=IDFLDAS,SM=(2,72,1)');
 IDFLDAS: PROC(DICT#);
 /*PROCEDURE TO GENERATE FLOWCHART RECORDS FOR IMPLICIT ASSOCIATIONS
 OF FIELDS OR INTERIM VARIABLES.  THE PARAMETER PASSED IS
 THE DICTIONARY ENTRY NUMBER OF THE FIELD OR INTERIM VARIABLE FOR WHICH
 WE ARE GENERATING A FLOWCHART ENTRY.  EVEN IF THE FIELD DOES NOT
 REQUIRE IMPLICIT ASSIGNMENT CODE TO BE GENERATED (I.E. IT HAS A VALUE
 BY VIRTUE OF AN ASSERTION OR  IT IS IN A SOURCE RECORD), THEN A
 FLOWCHART RECORD IS STILL GENERATED ANYWAY FOR REPORT PURPOSES.
 THUS, A FLOWCHART ENTRY WILL RESULT WHENEVER THIS ROUTINE IS INVOKED.
 THE TYPE INSERTED INTO THE NODE TYPE SHOWS WHAT IS TO BE DONE, AND
 IT IS DETERMINED AS FOLLOWS:
0          DICTYPE             ADJMAT TYPE        =>     NODE_TYPE
0          'FLD '              1                         'FLDS'
           'FLD '              3  OR 7                   'FLDP'
           'FLD '            . 4                         'FLDI'    *
          'INTR'               3  OR 7                   'INTP'
          'INTR'               4                         'INTI'    *
OTHE * INDICATES THAT CODE WILL BE GENERATED BY GIMFLD FOR THESE. */
-/* THIS PROCEDURE ALSO CALLS 'CHKATTR' IN ORDER TO CHECK THE ATTRIBUTES
 OF THE SOURCE AND TARGET VARIABLES FOR COMPATIBILITY, IN CASES
 WHERE CODE WILL BE GENERATED FOR IMPLICIT ASSIGNMENT  */
-DCL CHKATTR ENTRY(FIXED BIN, FIXED BIN), J FIXED BIN;
0/* WE CAN DECLARE MOST EXTERNALS VIA THE  %INCLUDE  */
 %INCLUDE INCLIB(DDICT);   /* DICTIONARY OF FULLY QUALIFIED NAMES. */


 DCL ADJMAT(*,*) FIXED BINARY CTL EXT;   /* ADJACENCY MATIRX. */
0/* THIS IS THE FORMAT OF THE FLOWCHART RECORD. */
 DCL 1 FLOWTAB_FIELD BASED(FLOWTAB_FIELD_PTR),
          2 NODE# FIXED BINARY,
          2 NODE_TYPE CHAR(4),
          2 TGT_FIELD CHAR(LEN_DICT_ENTRY),
          2 SRCE_FIELD CHAR(LEN_DICT_ENTRY),
     FLOWTAB FILE OUTPUT RECORD SEQUENTIAL;  /* THE FILE TO WHICH THE
 RECORD BELONGS. */
 DECLARE ENDING(7)CHAR(1)INITIAL('S',' ','P','I',' ',' ','P');
 DCL CODE FIXED BIN;
 DCL TYPE CHAR(3), DICT# FIXED BIN;
0/* THE TYPE IS SET TO THE FIRST 3 CHARACTERS OF THE TYPE AS INDICATED
      IN 'DICTYPE', WHICH SHOULD BE EITHER 'FLD' OR 'INT'  */
 TYPE=SUBSTR(DICTYPE(DICT#),1,3);
0/* FILL IN AS MUCH OF THE RECORD AS WE CAN. */
 LOCATE FLOWTAB_FIELD FILE(FLOWTAB);
 NODE#=DICT#;                                                              1
 TGT_FIELD= DICT(DICT#);
0/* CHECK THE APPROPRIATE COLUMN OF THE MATRIX FOR A NON-ZERO CODE,
    WHICH SIGNALS THE TYPE OF SOURCE THAT THE FIELD HAS, AS FOLLOWS:
       1 => FIELD IS IN A SOURCE RECORD;
       3 => FIELD HAS EXPLICIT SOURCE FROM AN ASSERTION;
       4 => FIELD HAS AN IMPLICIT SOURCE (ASSIGNMENT CODE WILL BE
                 NECESSARY); ANY OTHER TYPE IN ADJMAT IS IMPOSSIBLE  */
 /*  7=> FIELD HAS AN EXPLICIT CONDITIONAL SOURCE */
ODO J=1 TO HBOUND(ADJMAT,1);                                               2
      CODE=ADJMAT(J,DICT#);
      IF CODE¬=0 THEN DO;                                                  3
 /* FILL IN THE REST OF THE RECORD APPROPRIATLY. */
 /* USE THE CODE TO DECIDE WHAT TO DO. */
          NODE_TYPE=TYPE||ENDING(CODE);                                    6
          SRCE_FIELD=DICT(J);
 /* CASE 4 IS THE ONLY CASE WHERE PL/1 CODE WILL BE GENERATED. */
          IF CODE=4 THEN DO;
               CALL CHKATTR(DICT#,J);                                      8
               RETURN;
               END;
     IF (CODE=1)|(CODE=3)|(CODE=7) THEN RETURN;
                    ELSE CALL SYSERR('MODULE:IDLFDAS; BAD MAT CODE');
   END;
 END;
 /* NO CODE AT ALL SYSERR. */
 CALL SYSERR('MODULE: IDFLDAS; NO CODE ENCOUNTERED');
 END IDFLDAS;
```

```
*PROCESS('NST,MACRO,N=IDASSN,SM=(2,72,1),EXTREF');
 IDASSN: PROC(ASSN#);
 /* THIS PROCEDURE GENERATES A FLOWCHART ENTRY FOR AN ASSERTION. */
 DCL ASSN# FIXED BIN;
 %INCLUDE INCLIB(DDICT);
 %INCLUDE INCLIB(DSEDIR);
 %INCLUDE INCLIB(DASSTXT);
 %INCLUDE INCLIB(DASSNM);
 %DCL MAX#_COND_NODES FIXED;
 %DCL MAX_LEN_QNM   FIXED;
 %DCL MAX_LEN_LABEL  FIXED;
 % MAX#_COND_NODES=30;
 %MAX_LEN_LABEL=14;
 %MAX_LEN_QNM  =32;
 DCL  CONDAS(MAX#_COND_NODES)  CHAR(MAX_L_NAME) VAR EXT;
 DCL #CONDAS FIXED BIN EXT;   /* INDEX TO CONDAS TABLE  */
  /* TABLE OF NAMES OF ASSERTIONS THAT USE CONDITIONAL FUNCTIONS */
  /* DETECTED AND ENTERED BY THIS ROUTINE 'IDASSN', AND CHECKED BY
       IDREPL AND CHECOND   */
 DCL COND ENTRY(CHAR(*))RETURNS(BIT(1));
     DCL CHPTRLB  ENTRY(CHAR(*))  RETURNS(CHAR(MAX_LEN_QNM)VAR);
     DCL DICT# ENTRY(CHAR(*)VAR)RETURNS(FIXED BIN);
     DCL CONSONM ENTRY(FIXED BIN,FIXED BIN,POINTER)RETURNS
     (CHAR(MAX_LEN_QNM)VAR);
 DCL REPL_TARGET  CHAR(MAX_LEN_QNM)  VAR;
 DCL (FIXED_ASSERT INITIAL('$ASSERT'), SRCH_NAME) CHAR(MAX_L_NAME);
 DCL START EXT PTR;
 DCL ASSN_PTR(3) PTR;



 DCL CTR FIXED BIN;
 DCL (PRE_CHAR,CUR_CHAR) CHAR(1);
 DCL IN_QUOTE BIT(1);
 %DCL MAX_FN_NAME FIXED;
 % MAX_FN_NAME=7;
 DCL T_STRING CHAR(ST_LEN) VAR CTL;
 DCL J FIXED BIN;
 DCL INSRCF ENTRY(CHAR(*))  RETURNS(BIT(1));
 /* THIS IS THE RECORD WE ARE TO GENERATE. */
 DCL       1 FLOWTAB_ASSN BASED (FLOW_PTR),
                 2 NODE# FIXED BIN,
                 2 NODE_TYPE CHAR(4),              /* ASSN */
                 2 ASSERTION_NAME CHAR(MAX_L_NAME),
                 2 COND_FCN CHAR(MAX_FN_NAME),
                 2 REPLACE_LABEL  CHAR(5),
                 2 ALREADY_CALLED BIT(1),
                 2 LEN_TEXT FIXED BIN,
                 2 TEXT CHAR(N REFER (FLOWTAB_ASSN.LEN_TEXT));
0DCL FILEST ENTRY(CHAR(*)) RETURNS(CHAR(2));
 DCL TEMP_PLI_STR CHAR(47) VAR;
 DCL TEMP_SRC_FILE  CHAR(10) VAR;
0/* THIS IS ANOTHER LINE OF PLI TEXT THAT WE MAY PERHAPS GENERATE
      IF THER IS A NEED FOR A TEST OF SUBSET SELECTION */
0DCL 1 FLOWTAB_TEXT  BASED (FLOW_PTR),
       2 NODE# FIXED BIN,
       2 TYPE CHAR(4),       /* 'TEXT'  */
       2 LEN_TEXT FIXED BIN,
       2 PLI_STR  CHAR(N REFER(FLOWTAB_TEXT.LEN_TEXT));
0DCL ST_LEN FIXED BIN;
```

```
-/* WE USE RETREVE TO GET THE TEXT WE MUST CLEAN UP. */
 SRCH_NAME=DICT(ASSN#);
 CALL RETREVE(SRCH_NAME||'G'||FIXED_ASSERT,'ASTX',START,ASSN_PTR,CTR);    1
 IF CTR¬=1 THEN CALL SYSERR(  'IDASSN WRONG TEXT: '||SRCH_NAME);
 /* AS PER USUAL CHASE TO THE TEXT VIA THE POINTERS. */
 STORAGE_PTR=ASSN_PTR(1);
 DP=DATA_PT;                                                              2
 /* ALLOCATE THE TEMPORARY STRING WHERE THE MODIFIED TEXT WILL GO. */
 ST_LEN=LENTX;
 ALLOCATE T_STRING;
-/* WE WILL REMOVE EXTRANIOUS BLANKS. WE MUST BE CAREFUL ABOUT STRING
 CONSTANTS HOWEVER. */
 PRE_CHAR=' ';
 IN_QUOTE='0'B;
 T_STRING='';
 DO J=1 TO LENTX;
      CUR_CHAR=SUBSTR(STR,J,1);
      IF PRE_CHAR=' ' THEN DO;
            IF CUR_CHAR=' ' THEN DO;
                  IF IN_QUOTE THEN T_STRING=T_STRING||CUR_CHAR;
                  ELSE;
            END;
            ELSE T_STRING=T_STRING||CUR_CHAR;
      END;
      ELSE T_STRING=T_STRING||CUR_CHAR;
      IN_QUOTE=BOOL(IN_QUOTE,CUR_CHAR='''','0110'B);
 PRE_CHAR=CUR_CHAR;
 END;
 /* SET THE REFERING VARIABLE AND ALLOCATE THE RECORD. */
 N=LENGTH(T_STRING);
 LOCATE FLOWTAB_ASSN FILE(FLOWTAB);
 /* FILL THE RECORD. */
 TEXT=T_STRING;
 FREE T_STRING;


 FLOWTAB_ASSN.NODE#=ASSN#;                                                2
 FLOWTAB_ASSN.NODE_TYPE='ASSN';
 ASSERTION_NAME=SRCH_NAME;
-/* RETRIEVE THE ASSERTION TARGET STORAGE ENTRY (ASTG)
   FOR FURTHER ANALYSIS */
 CALL RETREVE(SRCH_NAME||'G'||FIXED_ASSERT,'ASTG', START,ASSN_PTR,CTR);
 IF CTR¬=1 THEN CALL SYSERR('IDASSN:ASTG NOT FOUND');
 STORAGE_PTR=ASSN_PTR(1);
 DP=DATA_PT;
0/* FIND OUT IF ASSERTION IS AN EXIST OR LEN TYPE ASSERTION OF A NAME IN
    IN A SOURCE FILE; THEN ALREADY CALLED BY BUFFER UNPACKING ROUTNS*/
0ALREADY_CALLED='0'B;
 DO I=2 TO #KEYS-2;
   IF NAME(I)='EXIST' | NAME(I)='LEN'  THEN
   DO;
   ALREADY_CALLED=INSRCF(NAME(I+1));
 END;
 END;
```

```
-/* FIND OUT WHETHER THE ASSERTION USES A SYSTEM CONDITIONAL FUNCTION OR
      THE "REPLACE" FUNCTION: */                                          3
 COND_FCN=FCN;      /* SETS THE FCN NAME IN THE ASSN FLOWTAB ENTRY */
-/* FIRST, CALL FCNUSED TO MARK THE FCN AS USED */
O CALL FCNUSED(FCN);                                                      4
O/* 'FCN' IS EITHER BLANK (NO FUNCTION USED BY THIS ASSERTION);
     OR  IS 'REPLACE' (THE SYSTEM FUNCTION BEING USED); OR IT IS THE
      NAME OF A SYSTEM-PROVIDED FUNCTION WHICH MAY BE CONDITIONAL */
-IF FCN='REPLACE' THEN  /* ASSERTION USES THE "REPLACE" FUNCTION  */    5
 DO;
        /* IF FUNCTION IS "REPLACEMENT"; IF SO, DETERMINE ITS REPLACEMENT
      VARIABLE  AND GENERATE THE LABEL IT WILL  BRANCH TO    IF REPLACE
       TOOK PLACE    */
O /* SET REPL TARGET VARIABLE; ASSUMPTION MADE HERE THAT AN ASSERTION
     USING THE REPLACE FUNCTION HAS ONLY ONE TARGET VARIABLE, WHICH IS IN
    THE 'ASTG' STORAGE ENTRY, STARTING AT POS 2 AND HAVING #COMPONENTS */
OREPL_TARGET=CONSONM(2,#COMPONENTS(1),STORAGE_PTR);
 J=DICT#(REPL_TARGET);
  PUT STRING(REPLACE_LABEL)     EDIT('$L',J)(A,P'999');
OEND;
-ELSE /* IF THE ASSERTION USES A FUNCTION & IT IS CONDITIONALLY-COMPLETE,
           I.E. IT IS IN THE 'SYSFCN' TABLE, THEN ENTER THE ASSERTION'S
           NAME IN THE 'CONDAS' (CONDITIONAL ASSERTIONS) TABLE */         6
O   IF FCN¬=' ' THEN
O      IF COND(FCN) THEN
       DO;
         #CONDAS=#CONDAS+1;
          IF #CONDAS>MAX#_COND_NODES THEN CALL SYSERR
                 ('ICASSN: TOO MANY CONDAS');
          CONDAS(#CONDAS)=CHPTPLB(ASSERTION_NAME);
       END;
-/* ALSO CHECK IF TARGET OF ASSERTION IS A 'SUBSET.X' TYPE NAME,
    MEANING THAT THE ASSERTION IS DESCRIBING A SUBSET OF A FILE X;
    IF SO, CHECK IF X IS A SOURCE FILE AND IF SO, WE NEED TO
    GENERATE CODE THAT IF THE SOURCE RECORD IS NOT IN THE SUBSET
    SPECIFIED, THEN   GO TO READ THE NEXT RECORD OF THAT FILE */
OLAST_POS_SUBSET=#KEYS-2;                                                 7
 DO K=2 TO LAST_POS_SUBSET;
    IF NAME(K)='SUBSET' THEN
    DO;
       /* CHECK IF CORRESPONDING FILE IS A SOURCE FILE */
       IF FILEST(NAME(K+1))='SR' THEN
       DO;
          TEMP_SRC_FILE=CHPTPL3(NAME(K+1));
          /* WE ARE GENERATING SPECIAL-PURPOSE CODE :



            "IF ¬SUBSET.X THEN GO TO $RD_XS;" WHERE X IS A FILENAME */
          TEMP_PL1_STR='IF ¬SUBSET.'||TEMP_SRC_FILE||' THEN GO TO $RD_'
                ||TEMP_SRC_FILE||'S;';
          N=LENGTH(TEMP_PL1_STR);
          LOCATE FLOWTAB_TEXT FILE( FLOWTAB);                             8
          FLOWTAB_TEXT.NODE#=0;
          FLOWTAB_TEXT.TYPE='TEXT';
          FLOWTAB_TEXT.PL1_STR=TEMP_PL1_STR;
       END;
    END;
 END;
-END ICASSN;
```

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=ICICCD');
 ICICCD: PROC(DICT_NO);
 %INCLUDE INCLIB(DDICT);
 %INCLUDE INCLIB(DANY);
 %INCLUDE INCLIB(DDISK);
 %INCLUDE INCLIB(DSEDIR);


 %INCLUDE INCLIB(DRECGRP);
 %DCL MAX_LEN_DRIVLAB FIXED;
 %MAX_LEN_DRIVLAB=4+MAX_L_NAME;
 %DCL MAX_LEN_QNM FIXED;
 %MAX_LEN_QNM=32;
 %DCL MAX_FLDS_RETR FIXED;
 %MAX_FLDS_RETR=200;
 DCL DICT_NO BIN FIXED(15);
 DCL 1 FTR BASED(FTR_PTR), /* FLOWTAB_REC */
         2 NODE# FIXED BIN,
         2 NODE_TYPE CHAR(4), /* 'RECD' */
         2 RECNAME CHAR(LEN_DICT_ENTRY),
         2 IOMODE CHAR(2),
                             /* RD FOR READ
                                WR FOR WRITE
                                RW FOR REWRITE */
         2 FILENAME CHAR(MAX_L_NAME),
         2 ORG CHAR(1),
                             /* S = SEQUENTIAL
                                I = INDEXED      */
         2 KEYED FIXED BIN,
                               /* 0 = NOT KEYED
                                  1 = KEYED      */
         2 KEYNAME CHAR(MAX_L_NAME),
         2 PACKED              BIN FIXED(15),
                                                   /* 0 = NOT PACKED
                                                      1 = PACKED      */
         2 RECORD_ARITY        BIN FIXED(15),
         2 #SUBSTRUCTURES      BIN FIXED(15),
         2 SUBSTRUCTURE(N REFER(FTR.#SUBSTRUCTURES)),
           3 NAME              CHAR(LENGTH_KEY_NAME),
           3 TYPE              CHAR(1),
                                                   /* F = FIELD
                                                      G = GROUP */
           3 #SUBSCRIPTS       BIN FIXED(15),
           3 SUBSCRIPT1        BIN FIXED(15),
           3 SUBSCRIPT2        BIN FIXED(15),
           3 EXIST_PROC        CHAR(LENGTH_KEY_NAME),
           3 ARITY             BIN FIXED(15),
           3 DATA_TYPE         CHAR(1),
                                                   /* C = CHARACTER
                                                      B = BINARY
                                                      N = NUMERIC
                                                      F = FIXED DECIMAL */
           3 FIELD_LEN_TYPE    CHAR(1),
                                                   /* F = FIXED
                                                      V = VARIABLE */
           3 MIN_LENGTH        BIN FIXED(15),
           3 MAX_LENGTH        BIN FIXED(15),
           3 LEN_PROC          CHAR(LENGTH_KEY_NAME);
 DCL FTR_PTR POINTER EXT;
-/* THIS IS THE TEMPORARY LOCATION OF INFORMATION WHICH WILL FIND ITS */
 /* WAY INTO THE UPPER PART OF FTR.                                   */
 DCL 1 UPPER,
         2 NODE#               BIN FIXED(15),
         2 NODE_TYPE           CHAR(4),
         2 RECNAME             CHAR(LEN_DICT_ENTRY),
         2 IOMODE              CHAR(2),
         2 FILENAME            CHAR(MAX_L_NAME),
         2 ORG                 CHAR(1),
         2 KEYED               BIN FIXED(15),
         2 PACKED              BIN FIXED(15),
         2 KEYNAME             CHAR(MAX_L_NAME),
              2 RECORD_ARITY          BIN FIXED(15),
              2 #SUBSTRUCTURES        BIN FIXED(15);
    0/* TEMP IS USED TO HOLD THE SUBSTRUCTURE ENTRIES WHICH FORM THE LAST */
     /* (LOWER) PART OF FTR.                                             */
     DCL 1 TEMP CTL,
              2 NAME                  CHAR(LENGTH_KEY_NAME),
              2 TYPE                  CHAR(1),
              2 #SUBSCRIPTS           BIN FIXED(15),
              2 SUBSCRIPT1            BIN FIXED(15),
              2 SUBSCRIPT2            BIN FIXED(15),
              2 EXIST_PROC            CHAR(LENGTH_KEY_NAME),
              2 ARITY                 BIN FIXED(15),
              2 DATA_TYPE             CHAR(1),
              2 FIELD_LEN_TYPE        CHAR(1),
              2 MIN_LENGTH            BIN FIXED(15),
              2 MAX_LENGTH            BIN FIXED(15),
              2 LEN_PROC              CHAR(LENGTH_KEY_NAME);
     DCL RECNAME_RETREVE CHAR(LENGTH_KEY_NAME);
     DCL START POINTER EXT;
     DCL STORAGE_PTR_LIST(MAX_FLDS_RETR) PTR;
     DCL #_STORAGE_ENTRIES FIXED BIN;
     DCL FILENAME_RETREVE CHAR(LENGTH_KEY_NAME);
     DCL BARE_FILENAME CHAR(LENGTH_KEY_NAME) VARYING;
     DCL STORAGE_DEVICE_NAME CHAR(LENGTH_KEY_NAME);
     DCL STORAGE_TYPE CHAR(4);
     DCL SOURCE_FILE BIT(1);
     DCL TARGET_FILE BIT(1);
     DCL SUFFIX CHAR(1); /* S OR T OR U */
     DCL NUMBER_PIC PIC'9999'; /* USED FOR SENDING NUMERIC INFO
                                 TO SYSERR. */
     DCL CHAR_FLAG CHAR(1); /* USED FOR RETURNING INFO FROM
                               FIELD_OR_GROUP.                 */
     DCL I              BIN FIXED(15);
     DCL IMEMBER        BIN FIXED(15);
     DCL $PFILE CHAR(LENGTH_KEY_NAME) EXT;
     DCL #TEMPS BIN FIXED(15);
     DCL CHPIRLS ENTRY(CHAR(*)) RETURNS(CHAR(LEN_DICT_ENTRY) VARYING);
     DCL SYSERR ENTRY(CHAR(*));
    0DCL DICT# ENTRY(CHAR(*) VARYING) RETURNS(BIN FIXED(15));
     /* RETURNS DICTIONARY NUMBER OF REQUESTED NAME, 0 IF NONE.          */
    0DCL 1 FLOWTAB_COND BASED(FLOWTAB_COND_PTR),
          2 NODE#      BIN FIXED(15),
          2 NODE_TYPE  CHAR(4), /* 'COND' */
          2 COND_NAME CHAR(MAX_LEN_CNM); /* NAME OF CONDITION */
    0DCL SUBSET_NAME CHAR(MAX_LEN_CNM) VARYING;                                 Algorithm IDIOCD
    0/* LABEL COMPOSED OF '$PC_'||FILENAME FOR DRIVING FILE;           */
     /* USED BY 'GETFLT' IN ORDER TO BRANCH BACK TO READ THE NEXT RECORD. */
     DCL DRIVLAB CHAR(MAX_LEN_DRIVLAB) VARYING EXTERNAL;
    0    CALL IDIOCD_CALLED; /* LABEL SYSPRINT THAT WE WERE CALLED. */
         UPPER.NODE#=DICT_NO;                                                     1
         UPPER.NODE_TYPE='RECD';                                                  2
         UPPER.RECNAME=DICT(DICT_NO);                                             3
    -/* DROP ANY PREFIX ('OLD.' OR 'NEW.') FROM                          */
     /* UPPER.RECNAME TO GET RECNAME_RETREVE, AN UNQUALIFIED RECORD      */
     /* NAME USED TO GET THE FILENAME, ETC.                              */
    0    IF (1 = INDEX(UPPER.RECNAME,'OLD.')) |
            (1 = INDEX(UPPER.RECNAME,'NEW.'))
         THEN RECNAME_RETREVE=SUBSTR(UPPER.RECNAME,5,10);                         4
         ELSE RECNAME_RETREVE=UPPER.RECNAME;
    -/* WHAT IS THE FILENAME?                                            */
     /* FILENAME_RETREVE WILL BE USED FOR RETREVE.                       */
     /* BARE_FILENAME WILL CONTAIN NO TRAILING BLANKS.                   */       5
    0    CALL RETREVE(RECNAME_RETREVE || '$$FILE     ' ||
                                       '|' ||
                                       RECNAME_RETREVE || '&$REPT      ',
                   '',
                   START,
                   STORAGE_PTR_LIST,
                   #_STORAGE_ENTRIES);
 0 IF #_STORAGE_ENTRIES ¬= 1
      THEN CALL SYSERR('IDIOCD:  NO FILE FOR RECORD ''' ||
                          RECNAME_RETREVE ||
                          '''');
 0     STORAGE_PTR=STORAGE_PTR_LIST(1);
 -/* WE'RE NOW LOOKING AT THE STORAGE ENTRY FOR A *FILE* STATEMENT.     */
 0     FILENAME_RETREVE=KEY_ENTRY(1).NAME;
       BARE_FILENAME=CHPTRL2(FILENAME_RETREVE);                              6
       UPPER.KEYNAME=KEY_ENTRY(4).NAME;
 0/* KEYNAME BLANK ==> UNKEYED                                          */
 0     IF UPPER.KEYNAME=' ' THEN UPPER.KEYED=0;
                           ELSE UPPER.KEYED=1;
       STORAGE_DEVICE_NAME=KEY_ENTRY(3).NAME;
  /* IF NOT SPECIFIED BY THE USER, THEN STORAGE_DEVICE_NAME WILL BE     */
  /* BLANK.                                                             */
  /* ASSUME:  CARDS     (SEQL) IF INPUT                                 */
  /*          PRINTER   (SEQL) IF OUTPUT                                */
 -/* FIND UPPER.ORG.                                                    */
  IF STORAGE_DEVICE_NAME=' ' /* UNSPECIFIED */
  THEN UPPER.ORG='S';
  ELSE DO;
  /* FIRST, GET THE STORAGE_TYPE:  CARD, PRNT, DISK, ETC.               */
 0     CALL RETREVE(STORAGE_DEVICE_NAME || '&¬$FILE      &¬$REPT      ',    7
                   '',
                   START,
                   STORAGE_PTR_LIST,
                   #_STORAGE_ENTRIES);
 0     IF #_STORAGE_ENTRIES ¬= 1
       THEN CALL SYSERR('IDIOCD:  NO STORAGE MEDIUM FOR ' ||
                          'STORAGE_DEVICE_NAME ''' ||
                          STORAGE_DEVICE_NAME ||
                          '''');
 0     STORAGE_PTR=STORAGE_PTR_LIST(1);
       DP=STORAGE_ENTRY.DATA_PT;
       STORAGE_TYPE=ANY_STMT.TYPE;
 -/* ANYTHING OTHER THAN DISK IS SEQUENTIAL.                            */
  /* IF IT'S DISK, CHECK THE DISK ORGANIZATION.                         */
 0     IF STORAGE_TYPE='DISK' THEN UPPER.ORG=DISK.ORGANIZATION;
                              ELSE UPPER.ORG='S';
      END;
 -/* IS THIS FILE USED AS *SOURCE* OR *TARGET* OR BOTH?                 */
  /* BOOLEAN VARIABLES "SOURCE_FILE" AND "TARGET_FILE" HAVE MEANINGS:   */
  /*            SOURCE_FILE    IFF   FILE IS A SOURCE                   */
  /*            TARGET_FILE    IFF      FILE IS A TARGET                */
 0/* IS IT A SOURCE?                                                    */
       CALL RETREVE(FILENAME_RETREVE,                                        8
                   'SRCF',
                   START,
                    STORAGE_PTR_LIST,
                    #_STORAGE_ENTRIES);
 0     IF #_STORAGE_ENTRIES ¬= 0 THEN SOURCE_FILE='1'B;
                                  ELSE SOURCE_FILE='0'B;
 0/* IS IT A TARGET?                                                    */
       CALL RETREVE(FILENAME_RETREVE,                                        8
                   'TARF',
                   START,
                    STORAGE_PTR_LIST,
                   #_STORAGE_ENTRIES);
   O     IF #_STORAGE_ENTRIES ¬= O THEN TARGET_FILE='1'B;
                                    ELSF TARGET_FILE='C'B;
   O/* ERROR IF NEITHER SCURCF NOP TARGET.                                  */
         IF (¬SCUPCE_FILE) & (¬TARGET_FILE)
         THEN CALL SYSERR('IDICCD:  FILE ''' ||
                          FILENAME_RETREVE ||
                          ''' IS NEITHER SCURCE NCR TARGET');
   -/* WHAT IS THE IOMODE?                                                  */     9
   O/* 'OLD.'     ==>  'RD';                                                */
    /* 'NEW.'     ==>  (SECL      ==>  'WR';                                */
    /*                  INDEXED   ==>  'RW');                               */
    /* NO PREFIX ==>  (SOUPCE_FILE ==> 'RC';                                */
    /*                  TAPGET_FILE ==> 'WR').                              */
   O     IF 1 = INDEX(UPPER.RECNAME,'CLD.')
         THEN UPPER.IOMOCE='RC';
         ELSE

   O     IF 1 = INDEX(UPPER.RECNAME,'NEW.')
         THEN IF UPPER.ORG='S' THEN UPPER.IOMCCE='WR';
                               ELSF UPPER.IOMCDE='RW';
         ELSE /* NO PREFIX */
              IF SOUPCE_FILF THEN UPPER.ICMODE='RD';
                             ELSE UPPER.ICMODE='WR';                                8
   -/* COMPUTE SUFFIX 'S' OR 'T' OR 'U' FOR FILENAMF.                       */
              IF UPPER.ORG='S'
              THEN IF UPPER.IOMODE='RC' THEN SUFFIX='S';
                                        ELSF SUFFIX='T';
         ELSE IF UPPER.ORG='I'
              THEN IF SOURCF_FILE & TARGET_FILE
                   THEN SUFFIX='U';
              ELSE IF SOURCF_FILE THEN SUFFIX='S';
                                  ELSE SUFFIX='T';
         ELSE /* UPPER.ORG IS NEITHER 'S' NOR 'I'.                          */
              CALL SYSERR('IDIDCD:  ILLEGAL ORG ''' ||
                          UPPER.ORG ||
                          ''' FOR RECNAME ''' ||
                          UPPER.RECNAME ||
                          '''');
         UPPER.FILENAME=BASE_FILENAME || SUFFIX;
   -/* DC WE NEED A FLOWTAB_CCND FOR A SUBSET SPECIFICATION?                */
    /* FIRST CF ALL, DETERMINE WHETHER GCING TC BE A WRITE OR REWRITE,      */
    /* AND IN THAT CASE WHETHER IT HAS A CORRESPONDING SUBSET SPECIFICA-    */
    /* TION.  THIS IS DCNE IN ORDER TC GENERATE CCNDITIONAL CODE TO TEST    */
    /* FOR WHETHER THE RECORD IS IN THE SUBSET SPECIFIED.  LOOK IN THE      */
    /* DICTIONARY TC SEE WHETHER FILE HAS A CORRESPONDING SUBSET SPEC.      */     10
   O     IF TARGET_FILE
         THEN DO;
              SUBSET_NAME='SUBSET.' || BASE_FILENAME;
              J=DICT#(SUBSET_NAME);
              IF J>0
              THEN DO;
                   LOCATE FLOWTAB_CCND FILE(FLCWTAB);
                   FLOWTAB_CCNC.NODE#=0;
                   FLOWTAB_CCNC.NODE_TYPE='CCNC';
                   FLOWTAB_CCNC.COND_NAME=SUBSET_NAME;
                   END;
              END;
   -/* SAVE LABEL TO DRIVING FILE, IF APPROPRIATE, AS 'DRIVLAB';           */
    /* WILL BE USED BY 'CENFLT' TC BRANCH BACK TO READ NEXT RECORD.         */
         IF UPPER.IOMODE='RD'
         THEN IF UPPER.ORG='S'
              THEN IF UPPER.KEYED=C
```

```
               THEN DRIVLAB='$RD_' || CHPTRLB(UPPER.FILENAME);
1/* DOES THIS RECORD REQUIRE PACKING/UNPACKING?                       */
0/* IF NEITHER *TAPE* NOR *DISK* NOR *TERM* THEN *NO*.               */
       IF (STORAGE_TYPE ¬= 'DISK')
          & (STORAGE_TYPE ¬= 'TAPE')                                      11
          & (STORAGE_TYPE ¬= 'TERM')
       THEN UPPER.PACKED=0;
       ELSE /* CHECK RECFM IN DATA_ENTRY.                              */
          /* NOTE:  WE'RE RELYING ON THE SIMILARITY OF THE FIRST 3    */
          /* ITEMS IN THE DATA_ENTRIES OF *DISK* AND *TAPE* AND        */
          /* *TERM*.  WATCH OUT IF THESE ARE CHANGED                   */
          IF DISK.RECFM > 1 /* NOT F OR FB */
          THEN UPPER.PACKED=1;
          ELSE UPPER.PACKED=0;
1/* IF NO PACKING/UNPACKING NEEDED, THEN WRITE FTR OUT TO THE FILE     */
 /* FLOWTAB.                                                           */
-      IF UPPER.PACKED=0
       THEN DO;
          /* FILL THE LAST ENTRIES OF UPPER. */
          UPPER.RECORD_ARITY=0;
          UPPER.#SUBSTRUCTURES=1; /* GRATUITOUS */
0/* LOCATE FTR AND CREATE A GRATUITOUS SUBSTRUCTURE ENTRY.             */
          N=1;
          LOCATE FTR FILE(FLOWTAB) SET(FTR_PTR);                          12
0/* MOVE DATA FROM UPPER TO FTR.                                       */
          CALL MOVE_UPPER_TO_FTR;
          GOTO FINISHED; /* FOR RETURN */
          END;
1/* IF WE REACH HERE, THEN THE RECORD NEEDS TO BE PACKED/UNPACKED.     */ Algorithm
-/* CREATE $PFILE FOR USE IN RETREVING FIELDS AND GROUPS.              */ IDPACK
     $PFILE='$P' || BARE_FILENAME;
-/* SET STORAGE_ENTRY AND DATA_ENTRY TO POINT TO THE RECORD UNDER      */
 /* CONSIDERATION.                                                     */
       CALL RETREVE(RECNAME_RETREVE || '&$RECD      ' || '|' ||
                                  RECNAME_RETREVE || '&$RPTN      ',
                  '',
                  START,
                  STORAGE_PTR_LIST,
                  #_STORAGE_ENTRIES);
0      IF #_STORAGE_ENTRIES ¬= 1
       THEN CALL SYSERR('IDIDCD:  ''' || RECNAME_RETREVE ||
                                  ''' DOES NOT OCCUR IN EXACTLY ONE ' ||
                                  'RECD/RPTN STMT');
-      STORAGE_PTR=STORAGE_PTR_LIST(1);
       DP=STORAGE_ENTRY.DATA_PT;
-/* RECORD_ARITY CAN BE FILLED IN IMMEDIATELY.                         */
       UPPER.RECORD_ARITY=RECGRP.#MEMBERS;
-/* CREATE TEMPS FOR EACH MEMBER, TRACING OUT THE STRUCTURE OF THE     */
 /* RECORD IN PREFIX.                                                  */
0/* INITIALIZE #TEMPS TO 0.                                            */
       #TEMPS=0;
       DO IMEMBER=1 TO UPPER.RECORD_ARITY;                                1'
          CALL CREATE_TEMP(STORAGE_ENTRY.NAME(IMEMBER+1),                 2
                           RECGRP.#SUB(IMEMBER),
                           RECGRP.FIRST_SUB(IMEMBER),
                           RECGRP.SECOND_SUB(IMEMBER));
       END;
-/* ON RETURN FROM CREATE_TEMP THERE WILL BE A STACK OF TEMPS, ONE     */
 /* FOR EACH NAME WITHIN THE RECORD STRUCTURE.                         */
-/* LOCATE FTR. */
       N=#TEMPS;
       UPPER.#SUBSTRUCTURES=#TEMPS;
       LOCATE FTR FILE(FLOWTAB) SET(FTR_PTR);
```

```
-/* FILL TOP PART OF FTR FROM UPPER.                                  */
      CALL MOVE_UPPER_TO_FTR;
-/* COPY THE TEMPS, IN REVERSE SEQUENCE, INTO FTR.SUBSTRUCTURE(*).     */
      DO I=#TEMPS TO 1 BY -1;
           CALL FILL_FTR_SUBSTRUCTURE(I);                                          3
           FREE TEMP;
      END;
 FINISHED:
 PUT EDIT('LEAVING IDIOCD')(SKIP,A);
 RETURN;
1IDIOCD_CALLED:  PROC;
-/* FOR DEBUGGING PURPOSES -                                           */
 /*    THIS PROCEDURE WRITES A LABEL TO SYSPRINT SAYING THAT IDIOCD   */
 /*     WAS CALLED.                                                   */
-     PUT EDIT('*******************')
           (SKIP(2),A);
0     PUT EDIT('* IDIOCD CALLED *')
           (SKIP,A);
0     PUT EDIT('*******************')
           (SKIP,A);
-RETURN;
 END; /* IDIOCD_CALLED */
1MOVE_UPPER_TO_FTR:  PROC;
-/* INTERNAL TO IDIOCD.                                                */
0/* THIS PROCEDURE MOVES INFORMATION FROM UPPER (A TEMPORARY DATA      */
 /* STRUCTURE) TO THE FINALLY ALLOCATED FTR.                          */
-     FTR.NODE#=UPPER.NODE#;
      FTR.NODE_TYPE=UPPER.NODE_TYPE;
      FTR.RECNAME=UPPER.RECNAME;
      FTR.IOMODE=UPPER.IOMODE;
      FTR.FILENAME=UPPER.FILENAME;
      FTR.ORG=UPPER.ORG;
      FTR.KEYED=UPPER.KEYED;
      FTR.PACKED=UPPER.PACKED;
      FTR.RECORD_ARITY=UPPER.RECORD_ARITY;
      FTR.#SUBSTRUCTURES=UPPER.#SUBSTRUCTURES;
0RETURN;
 END; /* MOVE_UPPER_TO_FTR */
1FILL_FTR_SUBSTRUCTURE:  PROC(ISTRUCT);
-/* INTERNAL TO IDIOCD.                                                */
0/* THIS PROCEDURE TRANSFERS INFORMATION FROM A TEMP STRUCTURE INTO    */
 /* THE FINALLY ALLOCATED FTR.                                        */
-DCL ISTRUCT BIN FIXED(15);
-     FTR.NAME(ISTRUCT)=TEMP.NAME;
      FTR.TYPE(ISTRUCT)=TEMP.TYPE;
      FTR.#SUBSCRIPTS(ISTRUCT)=TEMP.#SUBSCRIPTS;
      FTR.SUBSCRIPT1(ISTRUCT)=TEMP.SUBSCRIPT1;
      FTR.SUBSCRIPT2(ISTRUCT)=TEMP.SUBSCRIPT2;
      FTR.EXIST_PROC(ISTRUCT)=TEMP.EXIST_PROC;
      FTR.ARITY(ISTRUCT)=TEMP.ARITY;
      FTR.DATA_TYPE(ISTRUCT)=TEMP.DATA_TYPE;
      FTR.FIELD_LEN_TYPE(ISTRUCT)=TEMP.FIELD_LEN_TYPE;
      FTR.MIN_LENGTH(ISTRUCT)=TEMP.MIN_LENGTH;
      FTR.MAX_LENGTH(ISTRUCT)=TEMP.MAX_LENGTH;
      FTR.LEN_PROC(ISTRUCT)=TEMP.LEN_PROC;
-RETURN;
 END; /* FILL_FTR_SUBSTRUCTURE */
1CREATE_TEMP:  PROC(THINGNAME,#SUBS,SUB1,SUB2) RECURSIVE;
-/* INTERNAL TO IDIOCD.                                                */
0/* THINGNAME MAY BE THE NAME OF A GROUP OR FIELD.                     */
 /* #SUBS, SUB1, AND SUB2 DESCRIBE HOW MANY OF THIS THING APPEAR IN   */
 /* THE RECORD STRUCTURE; THEY'RE GOTTEN FROM THE NAME IMMEDIATELY    */
 /* ABOVE THIS ONE.                                                   */
```

```
-DCL THINGNAME CHAR(LENGTH_KEY_NAME);
 DCL (#SUBS,SUB1,SUB2) BIN FIXED(15);
-/* NOTE THAT                                                          */
 /*                   STORAGE_PTR                                     */
 /*                   DP                                              */
 /*                   STORAGE_PTR_LIST                                */
 /*                   #_STORAGE_ENTRIES                               */
 /* MUST BE LOCAL TO CREATE_TEMP IF IT IS TO BE RECURSIVE.            */
 DCL STORAGE_PTR          POINTER;
 DCL DP                   POINTER;
 DCL STORAGE_PTR_LIST(MAX_FLDS_PETR) PTR;
 DCL #_STORAGE_ENTRIES BIN FIXED(15);
 DCL 1 STORAGE_ENTRY BASED(STORAGE_PTR),
       2 #KEYS FIXED BINARY,
       2 DATA_PT POINTER,
       2 KEY_ENTRY(N REFER(#KEYS)),
         3 NAME CHAR(LENGTH_KEY_NAME),
         3 NEXT_PTR POINTER;
 %INCLUDE INCLIB(DFIELD);
```

Algorithm CREATE-TEMP

```
 /* INCLUDES DCL OF FIELD TEMPLATE
 %INCLUDE INCLIB(DRECGRP);
 /* INCLUDES DCL OF RECGRP                                            */
-/* IS THINGNAME A FIELD OR A GROUP?                                  */
 CALL FIELD_OR_GROUP(THINGNAME);
 IF CHAR_FLAG='F' /* FIELD */
 THEN CALL CREATE_TEMP_FOR_FIELD;
 ELSE CALL CREATE_TEMP_FOR_GROUP;
-RETURN;
```

2

```
1FIELD_OR_GROUP:  PROC(THINGNAME);
0/* INTERNAL TO CREATE_TEMP.                                          */
0/* THIS PROCEDURE TAKES A MEMBER NAME AND SETS CHAR_FLAG TO:         */
 /*      'F'  IF IT'S A FIELD                                         */
 /*      'G'  IF IT'S A GROUP                                         */
 /* SYSERR IS CALLLED IF NEITHER IS TRUE                              */
-DCL THINGNAME CHAR(LENGTH_KEY_NAME);
 DCL STORAGE_PTR_LIST(MAX_FLDS_PETR) PTR;
 DCL #_STORAGE_ENTRIES BIN FIXED(15);
 DCL DB_$PFILE CHAR(LENGTH_KEY_NAME);                                 DEBUG
 DB_$PFILE=$PFILE;                                                    DEBUG
-/* IS IT A FIELD?                                                    */
 CALL RETREVE(THINGNAME || '&' || $PFILE || '&$FLD        ',
              '',
              START,
              STORAGE_PTR_LIST,
              #_STORAGE_ENTRIES);
 IF #_STORAGE_ENTRIES = 1
 THEN DO;
      CHAR_FLAG='F'; /* FIELD */
      RETURN;
      END;
-/* IS IT A GROUP?                                                    */
 CALL RETREVE(THINGNAME || '&' || $PFILE || '&$GRP        ',
              '',
              START,
              STORAGE_PTR_LIST,
              #_STORAGE_ENTRIES);
 IF #_STORAGE_ENTRIES > 0
 THEN DO;
      CHAR_FLAG='G'; /* GROUP */
      RETURN;
      END;
-/* IF NEITHER FIELD NOR GROUP, THEN CALL SYSERR                      */
```

```
         CALL SYSERR('IDIOCD - FIELD_OR_GROUP:  ''' || THINGNAME ||
                     ''' IS NEITHER A GROUP NOR A FIELD');
    END; /* FIELD_OR_GROUP */
1CREATE_TEMP_FOR_FIELD:  PROC;
-/* INTERNAL TO CREATE_TEMP.                                               */
0/* THIS PROCEDURE ALLOCATES AND FILLS A COPY OF TEMP IN THE CASE        */
 /* WHERE THINGNAME IS THE NAME OF A FIELD.                                */
-      #TEMPS=#TEMPS+1;
       ALLOCATE TEMP;
-/* FILL TEMP WITH THE OBVIOUS INFORMATION FIRST                           */
       TEMP.NAME=THINGNAME;
       TEMP.TYPE='F'; /* FIELD */                                                     3
       TEMP.#SUBSCRIPTS=#SUBS; /* FROM CALLING ROUTINE */                            2
       TEMP.SUBSCRIPT1=SUB1;    /* FROM CALLING ROUTINE */
       TEMP.SUBSCRIPT2=SUB2;    /* FROM CALLING ROUTINE */
       TEMP.ARITY=0; /* A FIELD HAS NO MEMBERS */                                    4
-/* LOOK UP THE STORAGE_ENTRY AND DATA_ENTRY FOR THIS FIELD                */
       CALL RETREVE(THINGNAME || '&' || $PFILE,                                       5
                    'FLD ',
                    START,
                    STORAGE_PTR_LIST,
                    #_STORAGE_ENTRIES);
       STORAGE_PTR=STORAGE_PTR_LIST(1);
       DP=STORAGE_ENTRY.DATA_PT;
-/* FILL TEMP WITH INFORMATION FROM DATA_ENTRY.                            */
            IF FIELD_TYPE=0 THEN TEMP.DATA_TYPE='C'; /* CHARACTER */                  6
       ELSE IF FIELD_TYPE=1 THEN TEMP.DATA_TYPE='B'; /* BINARY */
       ELSE IF FIELD_TYPE=2 THEN TEMP.DATA_TYPE='N'; /* NUMERIC */
       ELSE IF FIELD_TYPE=3 THEN TEMP.DATA_TYPE='F'; /* FIXED-DECIMAL */
       ELSE DO;
            NUMBER_PIC=FIELD_TYPE;
            CALL SYSERR('IDIOCD:  BAD FIELD_TYPE IN FIELD DATA_ENTRY:  '
                              || NUMBER_PIC);
            END;
            IF FIELD.FIELD_LEN_TYPE=0 THEN TEMP.FIELD_LEN_TYPE='F';                  7
       ELSE IF FIELD.FIELD_LEN_TYPE=1 THEN TEMP.FIELD_LEN_TYPE='V';
       ELSE DO;
            NUMBER_PIC=FIELD.FIELD_LEN_TYPE;
            CALL SYSERR('IDIOCD:  BAD FIELD_LEN_TYPE IN FIELD ' ||
                              'DATA_ENTRY:  ' || NUMBER_PIC);
            END;
       TEMP.MIN_LENGTH=FIELD.FIELD_LEN_MIN;                                          8
       TEMP.MAX_LENGTH=FIELD.FIELD_LEN_MAX;
-/* RETREVE THE NAMES OF LEN AND EXIST ASSERTIONS IF NECESSARY.            */
-/* GET EXIST_PROC.                                                         */    9
       TEMP.EXIST_PROC=' ';
       IF (TEMP.#SUBSCRIPTS=2) & (UPPER.ICMODE='RD')
       THEN DO;
            CALL RETREVE(THINGNAME || '&EXIST    ',
                         'ASTG',
                         START,
                         STORAGE_PTR_LIST,
                         #_STORAGE_ENTRIES);
            IF #_STORAGE_ENTRIES ¬= 1
            THEN CALL SYSERR('IDIOCD:  EXIST_PROC MISSING FOR FIELD ''' ||
                             THINGNAME ||
                             '''');
            STORAGE_PTR=STORAGE_PTR_LIST(1);
            TEMP.EXIST_PROC=STORAGE_ENTRY.NAME(1);
            END;
-/* GET LEN_PROC.                                                           */
       TEMP.LEN_PROC=' ';
       IF (TEMP.FIELD_LEN_TYPE='V') & (UPPER.ICMODE='RD')                           10
```

```
         THEN DO;
              CALL RETREVE(THINGNAME || '&LEN        ',
                           'ASTG',
                           START,
                           STORAGE_PTR_LIST,
                           #_STORAGE_ENTRIES);
              IF #_STORAGE_ENTRIES ¬= 1
              THEN CALL SYSERR('IDIOCC:  LEN_PROC MISSING FOR FIELD ''' ||
                                THINGNAME ||
                                '''');
              STORAGE_PTR=STORAGE_PTR_LIST(1);
              TEMP.LEN_PROC=STORAGE_ENTRY.NAME(1);
              END;
 -RETURN;
  END; /* CREATE_TEMP_FOR_FIELD */
 1CREATE_TEMP_FOR_GROUP:  PROC;
 -/* INTERNAL TO IDIOCC.                                                  */
 0/* THIS PROCEDURE ALLOCATES AND FILLS COPIES OF TEMP FOR A GROUP.      */
 0DCL I                 BIN FIXED(15);
 -       #TEMPS=#TEMPS+1;
         ALLOCATE TEMP;
 -/* FILL TEMP WITH THE OBVIOUS INFORMATION FIRST.                        */
         TEMP.NAME=THINGNAME;
         TEMP.TYPE='G'; /* GROUP */
         TEMP.#SUBSCRIPTS=#SUBS;       /* FROM CALLING ROUTINE */                1
         TEMP.SUBSCRIPT1=SUB1;         /* FROM CALLING ROUTINE */
         TEMP.SUBSCRIPT2=SUB2;         /* FROM CALLING ROUTINE */
 -/* BLANK OUT THAT PORTION OF TEMP RELEVANT ONLY TO FIELDS.              */
         TEMP.DATA_TYPE=' ';
         TEMP.FIELD_LEN_TYPE=' ';
         TEMP.MIN_LENGTH=0;
         TEMP.MAX_LENGTH=0;
         TEMP.LEN_PROC=' ';
 -/* IS THERE AN EXIST_PROC?                                              */
         TEMP.EXIST_PROC=' ';                                                  12
         IF (TEMP.#SUBSCRIPTS=2) & (UPPER.ICMODE='RO')
         THEN DO;
              CALL RETREVE(THINGNAME || '&EXIST      ',
                           'ASTG',
                           START,
                           STORAGE_PTR_LIST,
                           #_STORAGE_ENTRIES);
              IF #_STORAGE_ENTRIES ¬= 1
              THEN CALL SYSERR('IDIOCC:  EXIST_PROC MISSING FOR GROUP ''' ||
                                THINGNAME ||
                                '''');
              STORAGE_PTR=STORAGE_PTR_LIST(1);
              TEMP.EXIST_PROC=STORAGE_ENTRY.NAME(1);
              END;
 -/* LOOK UP THE STORAGE_ENTRY AND DATA_ENTRY FOR THE STATEMENT:          */
  /*      "THINGNAME IS GROUP( . . . )".                                  */
         CALL GET_GROUP_STATEMENT;
 -/* FILL IN ARITY OF THE GROUP NAME.                                     */
         TEMP.ARITY=RECGRP.#MEMBERS;                                           13
 -/* RECURSIVELY CALL CREATE_TEMP, ONCE FOR EACH MEMBER OF THINGNAME.     */
         DO I=1 TO RECGRP.#MEMBERS;                                            14
              CALL CREATE_TEMP(STORAGE_ENTRY.NAME(I+1),
                               RECGRP.#SUB(I),
                               RECGRP.FIRST_SUB(I),
                               RECGRP.SECOND_SUB(I));
         END;
 -RETURN;
 1GET_GROUP_STATEMENT:  PROC;
```

```
-/* INTERNAL TO CREATE_TEMP_FOR_GROUP.                                */
0/* THIS PROCEDURE FINDS THE STORAGE_ENTRY AND DATA_ENTRY FOR           */
 /* THE STATEMENT:                                                    */
 /*        "THINGNAME IS GROUP( . . . );".                            */
0DCL I                 BIN FIXED(15);
0/* RETREVE ALL ENTRIES WITH                                          */
 /*       THINGNAME                                                   */
 /*       & $PFILE                                                    */
 /*       & GRP                                                       */
     CALL RETREVE(THINGNAME || '&' || $PFILE,
                  'GRP ',
                  START,
                  STORAGE_PTR_LIST,
                  #_STORAGE_ENTRIES);
-/* THERE ARE 3 CASES OF INTEREST:                                    */
 /* A.  1 ENTRY COMES BACK ==>                                        */
 /*           THINGNAME IS NOT A MEMBER OF ANY OTHER GROUP;           */
 /* B.  2 ENTRIES COME BACK ==>                                       */
 /*           THINGNAME IS A MEMBER OF ANOTHER GROUP.                 */
 /* C.  ELSE ==>  ERROR                                               */
-/* IF #_STORAGE_ENTRIES IS 1 THEN SET STORAGE_PTR AND DP AND RETURN. */
     IF #_STORAGE_ENTRIES = 1
     THEN DO:
          STORAGE_PTR=STORAGE_PTR_LIST(1);
          DP=STORAGE_ENTRY.DATA_PT;
          RETURN;
          END;
-/* IF #_STORAGE_ENTRIES IS 2, THEN FIND THE STORAGE_ENTRY WITH       */
 /* THINGNAME AS ITS FIRST KEY_NAME.                                  */
     IF #_STORAGE_ENTRIES = 2
     THEN DO;
          DO I=1 TO 2;
               STORAGE_PTR=STORAGE_PTR_LIST(I);
               IF STORAGE_ENTRY.NAME(1)=THINGNAME
               THEN DO; /* FOUND IT  */
                    DP=STORAGE_ENTRY.DATA_PT;
                    RETURN;
                    END;
          END;
0              /* HERE IFF NEITHER STORAGE_ENTRY HAS THINGNAME
                 AS ITS FIRST KEY_NAME */
               CALL SYSERR('ICICCD:  CAN''T FIND STORAGE_ENTRY FOR ' ||
                            'THE MODEL STATEMENT:  ''' || THINGNAME ||
                            ''' IS GROUP ( . . . );');
0         END;
-/* HERE IFF #_STORAGE_ENTRIES WAS NEITHER 1 NOR 2.                   */
               CALL SYSERR('ICICCD:  CAN''T FIND STORAGE_ENTRY FOR ' ||
                            'THE MODEL STATEMENT:  ''' || THINGNAME ||
                            ''' IS GROUP ( . . . );');
 END; /* GET_GROUP_STATEMENT */
 END; /* CREATE_TEMP_FOR_GROUP */
 END; /* CREATE_TEMP */
 END; /* ICICCD */
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=INDXGEN');
 INDXGEN:  PROC(N) RETURNS(CHAR(3));
0/* EXTERNAL TO GENICCD.                                              */
0/* THIS PROCEDURE GENERATES AN INDEX VARIABLE FOR THE INTEGER N.     */
0/* E.G.  INDXGEN(0)='I00'                                            */
 /*       INDXGEN(1)='I01'                                            */
0/* CURRENTLY, THE MAXIMUM INDEX ALLOWED IS N=27.                     */
0DCL N                 BIN FIXED(15);
 DCL RESULT            CHAR(3);
0  PUT STRING(RESULT) EDIT('I',N) (A,P'99');
```

```
      RETURN(RESULT);
 OEND; /* INDXCEN */
 *PRCCESS('MACRO,EXTREF,SM=(2,72,1),N=PACK');
  PACK:  PRCC(NEXT_INDFX) RECURSIVE;
  /* EXTERNAL TO GENERATE_PACKING.                                   */
  DCL NEXT_INDEX     BIN FIXED(15); /* THIS PARAMETER TELLS US WHICH  */
                                    /* IS THE NEXT_INDEX TO USE IN A  */
                                    /* CC-LCCP CR FCR SLBSCRIPTING.   */
  DCL NSTR BIN FIXED(15) EXT;
  DCL PTR POINTER;
  DCL PACKFLD ENTRY(BIN FIXEC(15));
  CCL PACKGRP ENTRY(BIN FIXED(15));
  DCL SYSERR ENTRY(CHAR(*));
  CCL FTR_PTR POINTER EXT;
  %CCL LEN_DICT_ENTRY FIXED: %LEN_DICT_ENTRY=32;
  %CCL MAX_L_NAME FIXED: %MAX_L_NAME=1C;
  %CCL LENGTH_KEY_NAME FIXED: %LENGTH_KEY_NAME=10;                    Algorithm
  %INCLUDE INCLIB3(FTR);                                               PACK
      PTR=FTR_PTR;
      NSTR=NSTR+1;
           IF TYPE(NSTR)='F' THEN CALL PACKFLC(NEXT_INDEX);             2
      ELSE IF TYPE(NSTR)='G' THEN CALL PACKGRP(NEXT_INCEX);
      ELSE /* TYPE(NSTR) ISN'T 'F' CR 'G'. */
           CALL SYSERR('GENICCD-PACK:  ILLEGAL TYPE ''' ||
                          TYPE(NSTR) ||
                          ''' FOR SUBSTRUCTURE NAMED ''' ||
                          NAME(NSTR) ||
                          '''');
  RETURN;
  END; /* PACK */
 *PROCESS('MACRO,EXTREF,SM=(2,72,1),N=PACKFLD');
  PACKFLD:  PROC(NEXT_INDEX);
  /* EXTERNAL TO PACK.                                               */
  /* GENERATES CODE FOR PACKING FIELCS.                              */
  /* THERE ARE 3 CASES OF INTEREST:                                  */
  /*                  #SUBSCRIPTS(NSTR) = 0                          */
  /*                  #SUBSCRIPTS(NSTR) = 1                          */
  /*                  #SUBSCRIPTS(NSTR) = 2                          */
  DCL NEXT_INDEX     BIN FIXED(15); /* THIS PARAMETER TELLS US WHICH  */
                                    /* IS THE NEXT_INDEX TO USE IN A  */
                                    /* CC-LCCP CR FCR SUBSCRIPTING.   */
  CCL NSTR BIN FIXED(15) EXT;
  CCL #RECNAME CHAR(32) VAR EXT;
  CCL #RECSTRING CHAR(34) VAR EXT;
  CCL PLISTR CHAR(320) VAR;
  DCL PTR POINTER;
  DCL SYSERR ENTRY(CHAR(*));
  DCL BYTE_CALC ENTRY(BIN FIXED(15),CHAR(1),BIN FIXEC(15));
  DCL SUBSCRIPT_STRING ENTRY RETURNS(CHAR(1CC) VAR);
  CCL FTR_PTR POINTER EXT;
  %DCL LEN_DICT_ENTRY FIXEC: %LEN_DICT_ENTRY=32;
  %DCL MAX_L_NAME FIXED: %MAX_L_NAME=10:
  %DCL LENGTH_KEY_NAME FIXED: %LENGTH_KEY_NAME=10;
  %INCLUDE INCLIB8(FTR);
  DCL PUSHSTK ENTRY(BIN FIXED(15));
  DCL POPSTK ENTRY;
  DCL #RPLI ENTRY(CHAR(*)VAR,CHAR(*));
  CCL INDXGEN ENTRY(BIN FIXED(15)) RETURNS(CHAR(3));
  DCL PICTURE ENTRY(BIN FIXED(15)) RETURNS(CHAR(1C)VAR);
  CCL CHPTRLB ENTRY(CHAR(*)) RETURNS(CHAR(32) VAR);
      PTR=FTR_PTR:
           IF #SUBSCRIPTS(NSTR) = C                                      3
           THEN DO;                                                      4
```

```
                        CALL GENFRATE_MCVE_INSTRS;
                        RETURN;
                        ENC;
     -     ELSE IF #SUBSCRIPTS(NSTR) = 1                                          5
                  THEN DO;
                        CALL PUSHSTK(NEXT_INCEX);
                        PLISTR='DO ' ||
                                INCXGEN(NEXT_INCEX) ||
                                '=1 TO ' ||
                                PICTURE(SUBSCRIPT1(NSTR)) ||
                                ';';
                        CALL WRPLI(PLISTR,'PLIEX');
                        CALL GENFRATE_MOVE_INSTRS;
                        PLISTR='ENC;';
                        CALL WRPLI(PLISTR,'PLIEX');
                        CALL POPSTK;
                        RETURN;
                        END;
     -     ELSE IF (#SUBSCRIPTS(NSTR)=2) & (SUBSCRIPT2(NSTR)=1)                   6
                  THEN DO:
                        PLISTR='DO ' ||
                                INOXGEN(NEXT_INCEX) ||
                                '=1 TC EXIST,' ||
                                CHPTRLB(NAME(NSTR)) ||
                                ';';
                        CALL WRPLI(PLISTR,'PLIEX');
                        CALL GENERATE_MCVE_INSTRS;
                        PLISTR='ENC;';
                        CALL WRPLI(PLISTR,'PLIEX');
                        RETURN;
                        END;
     -     ELSE IF #SUBSCRIPTS(NSTR) = 2                                          7
                  THEN DO;
                        CALL PUSHSTK(NEXT_INCEX);
                        PLISTR='DO ' ||
                                INCXGEN(NEXT_INCEX) ||
                                '=1 TC EXIST,' ||
                                CHPTRLB(NAME(NSTR)) ||
                                ';';
                        CALL WRPLI(PLISTR,'PLIEX');
                        CALL GENERATE_MCVE_INSTRS;
                        PLISTR='ENC;';
                        CALL WRPLI(PLISTR,'PLIEX');
                        CALL POPSTK;
                        RETURN;
                        END;
     -     ELSE /* #SUBSCRIPTS(NSTR) ISN'T C OR 1 CR 2.                          */
                  CALL SYSERR('GENIOCD - PACKFLC:  ILLEGAL #SUBSCRIPTS ' ||
                                PICTURE(#SUBSCRIPTS(NSTR)) ||
                                ' FCR SUPSTRUCTURE ''' ||
                                NAME(NSTR) ||
                                '''');                   Algorithm GEN-MOVE-INSTR-FOR-PACKING
    1GENERATE_MCVE_INSTRS:  PROC;
       /* INTERNAL TO PACK_FIELC.                                                */
       /* THIS PROCEDURE GENERATES THE PROPER INSTRUCTICNS FOR                   */
       /* CATENATING CHARACTER, BINARY, OR FIXED-DECIMAL INFORMATION             */
       /* CNTO THE OUTPUT STRING.                                                */
       DCL #BYTES BIN FIXED(15);
     -            IF DATA_TYPE(NSTR)='C'                                          1
                  THEN DO:
                        PLISTR=#RECSTRING ||                                      3
                                '=' ||
                                #RECSTRING ||
                              '||' ||
                              #RECNAME ||
                              '.' ||
                              CHPTRLB(NAME(NSTR)) ||
                              SUBSCRIPT_STRING ||
                              ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;
-      ELSE IF DATA_TYPE(NSTR)='B'                                              2
            THEN DO;                                                            5
                    CALL BYTE_CALC(MIN_LENGTH(NSTR),'B',#BYTES);
                    PLISTR=#RECSTRING ||
                              '=' ||
                              #RECSTRING ||
                              '||UNSPEC(' ||
                              #RECNAME ||
                              '.' ||
                              CHPTRLB(NAME(NSTR)) ||
                              SUBSCRIPT_STRING ||
                              ');';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;
-      ELSE IF DATA_TYPE(NSTR)='N'                                              3 ,
            THEN DO;
                    PLISTR=#RECSTRING ||
                              '=' ||
                              #RECSTRING ||
                              '||' ||
                              #RECNAME ||
                              '.' ||
                              CHPTRLB(NAME(NSTR)) ||
                              SUBSCRIPT_STRING ||
                              ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;                                                       5
-      ELSE IF DATA_TYPE(NSTR)='F'
            THEN DO;
                    CALL BYTE_CALC(MIN_LENGTH(NSTR),'B',#BYTES);
                    PLISTR=#RECSTRING ||
                              '=' ||
                              #RECSTRING ||
                              '||UNSPEC(' ||
                              #RECNAME ||
                              '.' ||
                              CHPTRLB(NAME(NSTR)) ||
                              SUBSCRIPT_STRING ||
                              ');';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;
-      ELSE /* DATA_TYPE(NSTR) ISN'T 'C' OR 'B' OR 'F' . */
            CALL SYSERR('GENIOCD - PACKFLD:  ILLEGAL DATA_TYPE ''' ||
                              DATA_TYPE(NSTR) ||
                              ''' IN SUBSTRUCTURE NAMED ''' ||
                              NAME(NSTR) ||
                              '''');
   END; /* GENERATE_MOVE_INSTRS */
   END; /* PACK_FIELD */
  *PROCESS('MACRO,EXTREF,SM=(2,72,1),N=PACKGRP');
   PACKGRP:  PROC(NEXT_INDEX) RECURSIVE;                                Algorithm PACKGRP
```

```
%CCL LEN_DICT_ENTRY FIXEC; %LEN_DICT_ENTRY=32;
%CCL MAX_L_NAME FIXED; %MAX_L_NAME=10;
%CCL LENGTH_KEY_NAME FIXED; %LENGTH_KEY_NAME=10;
/* EXTERNAL TO PACK.                                              */
/* GENERATES CODE FOR PACKING A GROUP AND CALLS PACK FOR EACH     */
/* MEMBER.                                                        */
/* THERE ARE 3 CASES OF INTEREST:                                 */
/*                 #SUBSCRIPTS(NSTR) = 0                          */
/*                 #SUBSCRIPTS(NSTR) = 1                          */
/*                 #SUBSCRIPTS(NSTR) = 2                          */
DCL NEXT_INDEX BIN FIXED(15);
DCL LOCAL_ARITY BIN FIXED(15);
CCL NSTR BIN FIXED(15) EXT;
DCL PLISTR CHAR(320) VAR;
DCL CHPTRLB ENTRY(CHAR(*)) RETURNS(CHAR(32)VAR);
DCL PACK ENTRY(BIN FIXED(15));
DCL PUSHSTK ENTRY(BIN FIXED(15));
CCL POPSTK ENTRY;
CCL INDXGEN ENTRY(BIN FIXED(15)) RETURNS(CHAR(3));
CCL PICTURE ENTRY(BIN FIXED(15)) RETURNS(CHAR(10)VAR);
CCL WRPLI ENTRY(CHAR(*)VAR,CHAR(*));
CCL SYSERR ENTRY(CHAR(*));
DCL FTR_PTR POINTER EXT;
%INCLUDE INCLIBB(FTR);
CCL I              BIN FIXED(15);
    PTR=FTR_PTR;
        IF #SUBSCRIPTS(NSTR) = 0                                          10
        THEN DO;
             LOCAL_ARITY=ARITY(NSTR);
             DO I=1 TO LOCAL_ARITY;
                  CALL PACK(NEXT_INDEX);
             END;
             RETURN;
             END;
    ELSE IF #SUBSCRIPTS(NSTR) = 1                                         11  (2)
        THEN DO;
             CALL PUSHSTK(NEXT_INDEX);
             PLISTR='DO ' ||
                     INDXGEN(NEXT_INDEX) ||
                     '=1 TO ' ||
                     PICTURE(SUBSCRIPT1(NSTR)) ||
                     ';';
             CALL WRPLI(PLISTR,'PLIEX');
             LOCAL_ARITY=ARITY(NSTR);
             DO I=1 TO LOCAL_ARITY;
                  CALL PACK(NEXT_INDEX+1);
             END;
             PLISTR='END;';
             CALL WRPLI(PLISTR,'PLIEX');
             CALL POPSTK;
             RETURN;
             END;
    ELSE IF (#SUBSCRIPTS(NSTR)=2) & (SUBSCRIPT2(NSTR)=1)                  11(3)
        THEN DO;
             PLISTR='DO ' ||
                     INDXGEN(NEXT_INDEX) ||
                     '=1 TO EXIST.' ||
                     CHPTRLB(NAME(NSTR)) ||
                     ';';
             CALL WRPLI(PLISTR,'PLIEX');
             LOCAL_ARITY=ARITY(NSTR);
             DO I=1 TO LOCAL_ARITY;
                  CALL PACK(NEXT_INDEX+1);
```

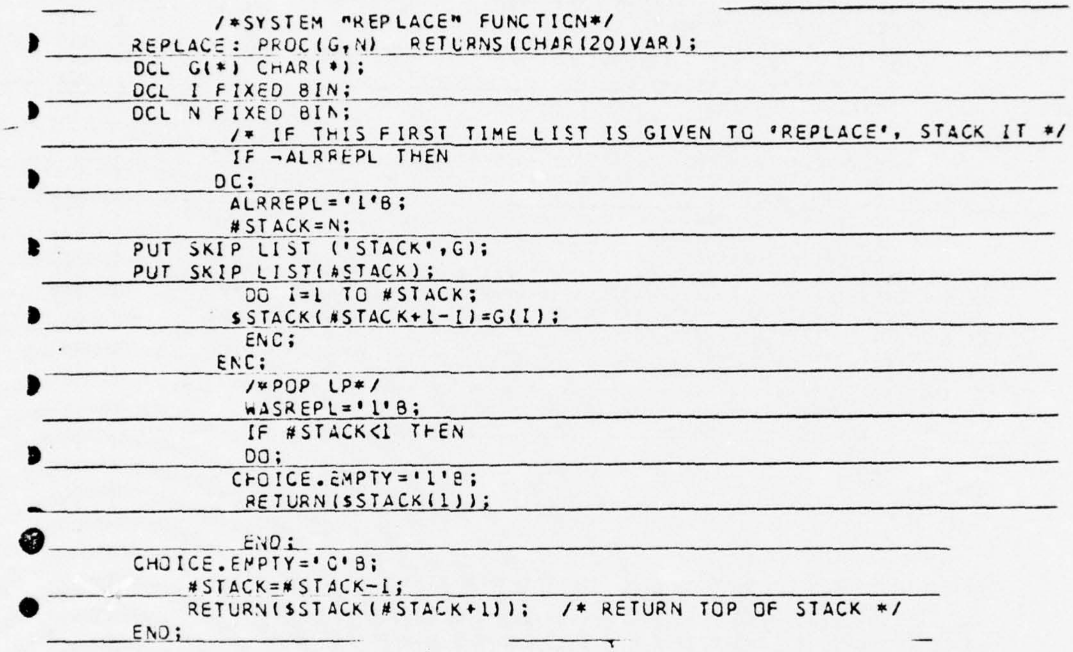
```
               END;
               PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIEX');
               RETURN;
               END;
-      ELSE IF #SUBSCRIPTS(NSTR) = 2                                        11 (4)
            THEN DO;
               CALL PUSHSTK(NEXT_INDEX);
               PLISTR='DO ' ||
                        INDXGEN(NEXT_INDEX) ||
                       '=1 TO EXIST.' ||
                        CHPTRLB(NAME(NSTR)) ||
                        ';';
               CALL WRPLI(PLISTR,'PLIEX');
               LOCAL_ARITY=ARITY(NSTR);
               DO I=1 TO LOCAL_ARITY;
                    CALL PACK(NEXT_INDEX+1);
               END;
               PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIEX');
               CALL POPSTK;
               RETURN;
               END;
-      ELSE /* #SUBSCRIPTS(NSTR) ISN'T 0 OR 1 OR 2. */
            CALL SYSERR('GENIOCD - PACKGRP:  ILLEGAL #SUBSCRIPTS ' ||
                        PICTURE(#SUBSCRIPTS(NSTR)) ||
                        ' FOR SUBSTRUCTURE NAMED ''' ||
                        NAME(NSTR) ||
                        '''');
   END; /* PACK_GROUP */
```

```
*PROCESS('NST,MACRO,N=IDGOTC');
 IDGOTC: PROC;
 %DCL MAX_LEN_LABEL  FIXED;
 %MAX_LEN_LABEL=14;
      DCL 1 FLOWTAB_PROTO BASED(P),
            2 NODE# FIXED BIN,
            2 TYPE CHAR(4),
            2 LABEL CHAR(MAX_LEN_LABEL);
      DCL DRIVLAB CHAR(MAX_LEN_LABEL)    VAR EXT;
      LOCATE FLOWTAB_PROTO  FILE(FLOWTAB);
            NODE#=0;
            TYPE='GOTO';
            LABEL=DRIVLAB;
 END IDGOTC;

*PROCESS('NST,MACRO,N=IDMODNM');
 IDMODNM: PROC(DICT#);


 /* THIS PROCEDURE CREATES A FLOWCHART RECORD FOR MODULE NAMES. */
 %INCLUDE INCLIB(DDICT);
 /* THE RECORD. */
 DCL 1 FLOWTAB_MODL BASED(FLOWTAB_MODL_PTR),
             2 NODE# FIXED BIN,
             2 NODE_TYPE CHAR(4),
             2 MODULE_NAME CHAR(10),
 DICT# FIXED BIN;
 /* CREATE RECORD. */
 LOCATE FLOWTAB_MODL FILE(FLOWTAB) SET(FLOWTAB_MODL_PTR);
 NODE#=DICT#;
 NODE_TYPE='MODL';
 MODULE_NAME=DICT(DICT#);
 RETURN;
 END IDMODNM;
```

```
*PROCESS('NST,SM=(2,72,1),N=IDRSET,MACRO');
 IDRSET: PROC;
 %INCLUDE INCLIB(DDICT);
 %DCL MAX_LEN_NAME FIXED;
 %DCL MAX_LEN_CNM   FIXED;
 %DCL MAX#_COND_NODES FIXED;
 %MAX_LEN_CNM  =32;
 %MAX#_COND_NODES=30;
 %MAX_LEN_NAME=10;
      DCL 1 FLOWTAB_RSET BASED(P),
           2 NODE# FIXED BIN,
           2 TYPE CHAR(4),
           2 NAME CHAR(MAX_LEN_CNM);
 %INCLUDE INCLIB(DSYSFCN);
      DCL CONDAS(MAX#_COND_NODES) CHAR(MAX_LEN_NAME) VAR EXT;
      /* THE 'CONDAS' (CONDITIONAL ASSERTIONS) TABLE CONTAINS THE NAMES
           OF ALL THE ASSERTIONS WHOSE COMPLETION IS CONDITIONAL; FILLED UP
           BY 'IDASSN'  */
   DCL #CONDAS FIXED BIN EXT;
0/* RESET  ALL RUN-TIME CONDITIONAL FLAGS  */
      DO K=1 TO #CONDAS;
           LOCATE FLOWTAB_RSET FILE(FLOWTAB);
           FLOWTAB_RSET.NODE#=0;
           FLOWTAB_RSET.TYPE='RSET';
           FLOWTAB_RSET.NAME=CONDAS(K) ||'_COMPLETED';
 END;
-/* IF THE REPLACE FUNCTION WAS EVER USED, RESET THE RUN-TIME FLAGS
    'WASREPL','FIRSTREPL' & THE STACK INDEX '#STACK'*/
 IF USEFCN(1) THEN  /* USED THE 'REPLACE' FCN  */
 DO;
           LOCATE FLOWTAB_RSET FILE(FLOWTAB);
           FLOWTAB_RSET.NODE#=0;
           FLOWTAB_RSET.TYPE='RSET';
      FLOWTAB_RSET.NAME='WASREPL';
           LOCATE FLOWTAB_RSET FILE(FLOWTAB);
           FLOWTAB_RSET.NODE#=0;
           FLOWTAB_RSET.TYPE='RSET';
      FLOWTAB_RSET.NAME='ALRREPL';
           LOCATE FLOWTAB_RSET FILE(FLOWTAB);
           FLOWTAB_RSET.NODE#=0;
           FLOWTAB_RSET.TYPE='RSET';
      FLOWTAB_RSET.NAME='#STACK';
-/* ALSO RESET ALL CHOICE NAMES  */
0DO J=1 TO DICTINO WHILE(DICT(J)<'CHOICE.ZZZZZ');
     IF LENGTH(DICT(J))    >7  THEN IF SUBSTR(DICT(J),1,7)='CHOICE.'
      THEN DO;
                   LOCATE FLOWTAB_RSET FILE(FLOWTAB);


                   FLOWTAP_RSET.NODE#=0;
                   FLOWTAB_RSET.TYPE='RSET';
                   FLOWTAB_RSET.NAME=DICT(J);
              END;
 END;
-END IDRSET;
```

```
*PROCESS('NST,SM=(2,72,1),N=MERGPL1,MACRO');
 MERGPL1: PROC;
 /* THIS PROCEDURE MERGES THE PL1_CODE FILES INTO ONE PL1PROG FILE*/
 DCL PL1STR CHAR(80) BASED(P);
 DCL(PL1DCL_EOF,PL1ON_EOF,PL1EX_EOF,PL1PROC_EOF) BIT(1)INIT('0'B);
 DCL TEXT_EOF BIT(1) INIT('0'B);
 DCL FILE_TEXT_NAME CHAR(10) VAR;
 DCL RUNTEXT_EOF BIT(1) INIT('0'B);
 OPEN FILE(PL1PROG)OUTPUT RECORD SEQL,
       FILE(PL1DCL)INPUT RECORD SEQL,
       FILE(PL1ON) INPUT RECORD SEQL,
       FILE(PL1EX) INPUT RECORD SEQL,
       FILE(PL1PROC) INPUT RECORD SEQL;
 %INCLUDE INCLIB(DSYSFCN);
-/*COPY DECLARATIONS FILE*/
 ON ENDFILE(PL1DCL)PL1DCL_EOF='1'B;
 READ FILE(PL1DCL)SET(P);
 DO WHILE(¬PL1DCL_EOF);
      WRITE FILE(PL1PROG) FROM(PL1STR);
      READ FILE(PL1DCL) SET(P);
 END;
-/* COPY PL1ON FILE*/
 ON ENDFILE(PL1ON) PL1ON_EOF='1'B;
 READ FILE(PL1ON) SET(P);
 DO WHILE(¬PL1ON_EOF);
      WRITE FILE(PL1PROG) FROM(PL1STR);
      READ FILE(PL1ON) SET(P);
 END;
 /*COPY PL1EX FILE*/
 ON ENDFILE(PL1EX) PL1EX_EOF='1'B;
 READ FILE(PL1EX) SET(P);
 DO WHILE(¬PL1EX_EOF);
      WRITE FILE(PL1PROG) FROM(PL1STR);
      READ FILE(PL1EX) SET(P);
 END;
 DO I=1 TO #SYSFCN;
 IF USEFCN (I) THEN DO;
      FILE_TEXT_NAME =FCNTEXT(I);
      TEXT_EOF ='0'B;



      OPEN FILE(TEXT)INPUT RECORD SEQL TITLE(FILE_TEXT_NAME);
  ON ENDFILE(TEXT)TEXT_EOF='1'B;
 READ FILE(TEXT) SET(P);
 DO WHILE(¬TEXT_EOF);
      WRITE FILE(PL1PROG) FROM (PL1STR);
      READ FILE(TEXT) SET(P);
 END;
 END;
 CLOSE FILE(TEXT);
 END;
-/* COPY OTHER RUNTIME ROUTINES */
 ON ENDFILE(RUNTEXT)    RUNTEXT_EOF ='1'B;
    READ FILE(RUNTEXT)  SET(P);
   DO WHILE( ¬RUNTEXT_EOF);
       WRITE FILE(PL1PROG) FROM(PL1STR);
       READ FILE(RUNTEXT)  SET(P);
   END;
-/*COPY PL1PROC FILE*/
 ON ENDFILE(PL1PROC)PL1PROC_EOF='1'B;
 READ FILE(PL1PROC) SET(P);
 DO WHILE(¬PL1PROC_EOF):
      WRITE FILE(PL1PROG) FROM(PL1STR);
      READ FILE(PL1PROC) SET(P);
 END;
0CLOSE FILE(PL1PROG);
-END;
```

```
*PROCESS('NST,SM=(2,72,1),N=PRECED');
  PRECED:PROC(ADJMAT, ORDER, N);
  /* REARRANGE NODES OF A DIRECT GRAPH(SPECIFIED BY ADJACENCY MATRIX   */PREC  01
  /* "ADJMAT",N BY N) IN ASCENDING "RANK" ORDER, RESULTING IN N-ELEMENT*/
  /* VECTOR "ORDER".                                                    */
    DCL (ADJMAT(*,*), UNUSE(N), T INIT('1'B), F INIT('0'B) ) BIT(1),
        (ORDER(*),NODES(2)INIT((2)0),NEW INIT(2),OLD INIT(1))FIXED BIN;
    DCL (          DEPTH(2,N), K INIT(0) ) FIXED BIN;
    DCL RANK(NR) FIXED BIN EXT CTL;
    DCL (I,J,L,II,M) FIXED BIN STATIC;


  /* ADJMAT--ADJACENCY MATRIX DEFINING THE DIGRAPH.                    */
  /* N-------THE NUMBER OF NODES IN THE DIGRAPH.                       */
  /* ORDER---THE VECTOR OF NODES IN RANK ORDER.                        */
  /* RANK----VECTOR OF RANKS OF NODES IN DIGRAPH.    INTUITIVELY, THE  */
  /*         RANK IS THE MAX. DEPTH OF A GIVEN NODE IN THE DIGRAPH     */
  /*         FROM ANY ROOT. EXT BECAUSE USED IN 'GFLTRPT'              */
  /* DEPTH---SET OF NODES IN A GIVEN RANK(OLD & NEW), I.E., "RANK SET".*/
  /* NODES---COUNTERS FOR # OF NODES IN OLD & NEW RANK SET(DEPTH).     */
  /* OLD-----POINTER TO PREVIOUS RANK SET.                             */
  /* NEW-----POINTER TO CURRENT RANK SET.                              */
  /* UNUSE---BIT VECTOR OF NODES NOT IN THE CURRENT RANK SET.          */
0/* ALLOCATE RANK AND INITIALIZE RANK OF ALL NODES TO 0  */                8
0     NR=N;
      ALLOCATE RANK;
0     RANK, ORDER = 0;                                                    9   1
    /*  SET UP DEPTH 0.  */                                              10
      DO J=1 TO N;                                                       11
       /* ENTER THOSE NODES WITH AN ALL-0 COLUMN IN THE FIRST RANK SET */
         IF ANY(ADJMAT(*,J)) THEN GOTO OUT;                              12   2
         M, NODES(OLD)=NODES(OLD)+1;    DEPTH(OLD,M)=J;                  13
      OUT: END;                                                          14
0     IF NODES(OLD)<=0 THEN GOTO ERR;     /* NO COL HAS ALL 0 */         15
                                       /* WHICH MEANS THAT EVERYTHING IS
                                           DEPENDENT ON SOMETHING ELSE */
0  /* OTHERWISE, PROCEED TO FIND RANK SETS OF DEPTH 1 AND ON */
0     DO L=1 TO N-1;     /* FOR EACH RANK SET ("DEPTH")  */              17  3
        /* INITIALIZE NUMBER OF NODES IN NEXT RANK SET TO 0;                 4
           FLAG ALL NODES AS NOT APPEARING IN NEXT RANK SET INITIALLY */
       NODES(NEW)=0;    UNUSE=T;                                         18
0      DO I=1 TO NODES(OLD);    II=DEPTH(OLD,I);  /* FOR EACH NODE IN THE     5
                                                   PREVIOUS RANK SET */
0        DO J=1 TO N;     /* FOR EACH COLUMN (NODE) CHECK IF IT IS A
                                DEPENDENT OF NODE IN OLD RANK SET  */
0          /* IF NODE IS DEPENDENT OF CURRENT NODE IN OLD RANK SET
                (NODE II)  AND IF IT IS NOT YET IN NEXT (NEW) RANK SET
                THEN ENTER IT AS A MEMBER OF THE NEXT RANK SET */
           IF ADJMAT(II,J) THEN IF UNUSE(J) THEN
              DO; RANK(J)=L:   UNUSE(J)=F; /* SET OR UPDATE RANK OF NODE      6
                                  AND INDICATE THAT IT IS NEW RANK SET*/
                 M, NODES(NEW)=NODES(NEW)+1;    DEPTH(NEW,M)=J;
              END;
         END;
       END;
0      /* IF THERE ARE NOT ANY NODES IN NEXT RANK SET, I.E. THERE
              ARE NO NODES DEPENDENT ON ANY NODES IN PREVIOUS RANK SET,
              THEN WE ARE DONE BECAUSE EVERY NODE HAS ITS RANK */
       IF NODES(NEW)<=0 THEN GOTO RECHDER;                                     7
0      /* EXCHANGE OLD AND NEW RANK SETS, WHICH HAS THE EFFECT OF
           MAKING NEW RANK SET THE OLD ONE, AND A NEW "NEW" RANK SET
           WILL BE CREATED IN NEXT PASS */
       M=NEW;   NEW=OLD;   OLD=M;
     END;
 -  ERR:
       RETURN;  /*CYCLES EXIST.  ERROR RETURN WITH ORDER=0 */
 -  RECHDER:  /* SORT NODES BY ASCENDING RANK ORDER.  */                       8
     DO I=0 TO L-1;
      DO J=1 TO N;
       IF RANK(J)=I THEN DO;  K=K+1;   ORDER(K)=J;  END;                 34
  END PRECED;
```



```
            /*SYSTEM "REPLACE" FUNCTICN*/
      REPLACE: PROC(G,N)  RETURNS(CHAR(20)VAR);
      DCL G(*) CHAR(*);
      DCL I FIXED BIN;
      DCL N FIXED BIN;
             /* IF THIS FIRST TIME LIST IS GIVEN TO 'REPLACE', STACK IT */
             IF ¬ALRREPL THEN
            DC;
             ALRREPL='1'B;
             #STACK=N;
      PUT SKIP LIST ('STACK',G);
      PUT SKIP LIST(#STACK);
              DO I=1 TO #STACK;
             $STACK(#STACK+1-I)=G(I);
              ENC;
            ENC;
              /*POP UP*/
              WASREPL='1'B;
              IF #STACK<1 TFEN
              DO;
             CFOICE.EMPTY='1'B;
              RETURN($STACK(1));

              END;
      CHOICE.EMPTY='C'B;
          #STACK=#STACK-1;
          RETURN($STACK(#STACK+1));  /* RETURN TOP OF STACK */
      END;
```

```
*PROCESS('MACRO,NST,EXTREF,N=RETREVE');
 RETREVE:PROCEDURE(LOGICAL,CONTROL,START,RETPTRS,M);
 /* LOGICAL IS A STRING OF NAMES UNITED BY LOGICAL EXPRESSIONS */        1  116
 /* CONTUL IS A STRING USED FOR CHECKING THE DATA */                      1  116
 /* START GIVES THE FIRST DIRECTORY_ENTRY */                              1  116
 /* RETPTRS IS THE  ARRAY OF POINTERS SATISFYING THE REQUEST (TO BE FILLED
    BY 'RETREVE')  */
 /* M IS THE NUMBER OF ENTRIES IN THE RETPTRS ARRAY; I.E. THE NUMBER
     OF ENTRIES SATISFYING THE REQUEST  */
          DCL (LOGICAL,CONTROL) CHAR(*);                                   1  116
          %DCL MAX_LENGTH_DATASTR FIXED;                                   1  118
          %DCL MAX_LEN_ARRAY FIXED;                                        1  119
          %MAX_LENGTH_DATASTR=100;                                         1  121
          %MAX_LEN_ARRAY=100;                                              1  122
          DCL RETPTRS(*) POINTER;
          DCL M FIXED BIN;
          DCL WORKARRAY(LEN_WORK_ARRAY) POINTER CTL;
 /* WOKARRAY IS USED FOR TEMPORARY STORAGE */                             1  126
          DCL SIGN BIT(1);                                                 1  126
          DCL START POINTER;                                               1  127
       %INCLUDE INCLIB(DSEDIR);
          DCL (KK,MM) FIXED BINARY;                                        1  130
 /* KK INDICATES THE CURRENT NAME IN LOGICAL WHICH IS PROCESSED */       1  131
 /* MM GIVES THE NUMBER OF LOCATIONS IN WORKARRAY THAT ARE FULL */       1  131
          DCL ITEM CHAR(LENGTH_KEY_NAME);                                  1  131
 /* ITEM IS THE KEY_NAME ANALYSED */                                      1  132
          DCL SYMBOL CHAR(1);                                              1  132
 /* SYMBOL IS USED FOR CHECKING LOGICAL OPERATORS */                      1  133
 /* ALLOCATE 'WORKARRAY' WITH THE SAME NUMBER OF ENTRIES AS RETPTRS */
          LEN_WORK_ARRAY=HBOUND(RETPTRS,1);
          ALLOCATE WORKARRAY;
          MM=0;                                                            1  133
          KK=1;                                                            1  134
 GET_NAME:                                                                 1  135    1
          ITEM=SUBSTR(LOGICAL,KK,LENGTH_KEY_NAME);                         1  135


 /*THE FIRST KEY DOESN'T HAVE A ¬ IN FRONT OF IT */                       1  136
          CALL RETRSE(ITEM,CONTROL);                                       1  136   2- 7
 /*RETRSE RETRIEVES THE STORAGE ENTRIES WHICH CONTAIN THE GIVEN           1  137
 KEY. THEY ARE PUT IN RETPTRS AND M IS  IS THE NUMBER OF SUCH             1  137
 STORAGE_ENTRIES */                                                        1  137
          KK=KK+LENGTH_KEY_NAME;                                           1  137    8
 TEST_LENGTH:                                                              2  138
             IF(KK>=LENGTH(LOGICAL))THEN                                   2  138 )
             GO TO EXIT;                                                   2  139
          SYMBOL=SUBSTR(LOGICAL,KK,1);                                     1  140    9
             IF(SYMBOL¬='&')THEN                                           2  141   10
             GO TO OR;                                                     2  142
          SYMBOL=SUBSTR(LOGICAL,KK+1,1);                                   1  143  14
             IF(SYMBOL='¬')THEN                                            2  144   15
             GO TO NOT;                                                    2  145
          SIGN='0'B;                                                       1  146
          ITEM=SUBSTR(LOGICAL,KK+1,LENGTH_KEY_NAME);                       1  147
          KK=KK+LENGTH_KEY_NAME+1;                                         1  148
 CONTINUE:                                                                 1  149
          CALL CHECK_FOR_KEY(ITEM,SIGN);                                   1  149   16 ,
 /* CHECK_FOR_KEY TAKES EVERY NON-NULL POINTER IN RETPTRS AND CHECKS      1  150
 IF THE CORRESPONDENT STORAGE_ENTRY HAS THE NAME INDICATED BY  ITEM       1  150   18
 THEN IT CHECKS  IF THE SIGN IS CORRECT. IF NOT THE POINTER IS SET TO     1  150
 NULL, OTHERWISE LET UNCHANGED */                                          1  150
          GO TO TEST_LENGTH;                                               1  150    19
 NOT:     SIGN='1'B;                                                       1  151
          ITEM=SUBSTR(LOGICAL,KK+2,LENGTH_KEY_NAME);                       1  152
          KK=KK+LENGTH_KEY_NAME+2;                                         1  153
          GO TO CONTINUE;                                                  1  154
 OR:      CALL MERGE;                                                      1  155   12  -
 /*MERGE CHECKS IF THE NON-NULL POINTERS IN RETPTRS ARE IN WORKARRAY.    1  156
 IF NOT THEY ARE ADDED IN THE ORDER IN WHICH THEY APPEAR IN RETPTRS       1  156
 OTHERWISE NOTHING HAPPENS */                                              1  156
          KK=KK+1;                                                         1  156
          GO TO GET_NAME;                                                  1  157    13
 EXIT:    CALL MERGE;                                                      1  158
          M=MM;                                                            1  159
 /* HERE THE CONTENT OF WORKARRAY IS PUT IN RETPTRS AND M SET TO MM */    2  160
              DO KK=1 TO MM;                                               2  160   20
              RETPTRS(KK)=WORKARRAY(KK);                                   2  161
              END;                                                         2  162
    FREE WORKARRAY;
```

```
1RETRSE:     PROCEDURE(KEY,DATAST);                                   2  163   2 - 7
 /* "RETRSE" (RETRIEVE STORAGE ENTRY) RETRIEVES ALL THE STORAGE       2  164
 ENTRIES WITH A KEY NAME "KEY" AND AUXILLARY DATA="DATAST".           2  164
 IF "DATAST"='' THEN THE AUXILLIARY DATA IS NOT CHECKED.              2  164
 THE POINTERS TO THE STORAGE ENTRIES WITH "KEY" ARE PLACED IN AN      2  164
 ARRAY OF POINTERS "RETPTRS" WHILE THE NUMBER OF POINTERS IS "M" */   2  164
            DCL DATAST CHAR(*);                                       2  164
            DCL KEY CHAR(LENGTH_KEY_NAME);                            2  165
            DCL LEN FIXED BINARY;                                     2  166
            DCL DATA_PTR POINTER;                                     2  167
            DCL SCANST ENTRY(POINTER) RETURNS (POINTER);              2  168
            DCL DATASTR CHAR(MAX_LENGTH_DATASTR) BASED (DATA_PTR);    2  169
            DCL NULL BUILTIN;                                         2  170
            M=0;                                                      2  171
            CALL CHECKDIRET;                                          2  172
               IF DIRECTORY_PTR¬=NULL THEN                            3  173
                  DO;                                                 4  174
                  STORAGE_PTR=FIRST_IN_LIST;                          4  175
                  LEN=LENGTH(DATAST);                                 4  176
                     IF LEN=0 THEN                                    5  177
                        DO;                                           6  178


                           DO WHILE(STORAGE_PTR¬=NULL);               7  179
                           CALL ENT_RETPTRS;
                           STORAGE_PTR=SCANST(STORAGE_PTR);           7  182
                           END;                                       7  183
                        END;                                          6  184
                     ELSE                                             5  185
                        DO;                                           6  185
                           DO WHILE(STORAGE_PTR¬=NULL);               7  186
                           DATA_PTR=DATA_PT;                          7  187
                              IF DATAST=SUBSTR(DATASTR,1,LEN)THEN     8  186
                               CALL ENT_RETPTRS;
                           STORAGE_PTR=SCANST(STORAGE_PTR);           7  193
                           END;                                       7  194
                        END;                                          6  195
                  END;                                                4  196
 0ENT_RETPTRS: PROC;
      M=M+1;                                                                 Step 5
      IF M>HBOUND(RETPTRS,1) THEN CALL SYSERR('MAX RETRIEVALS EXCEEDED');
      ELSE RETPTRS(M)=STORAGE_PTR;
  END ENT_RETPTRS;
 0CHECKDIRET:    PROCEDURE;                                           3  197   Step 2
  /* THIS PROCEDURE LOOKS FOR "KEY_NAME" IN THE DIRECTORY AND         3  198
  SETS THE "DIRECTORY_PTR" TO IT */                                   3  198
               DIRECTORY_PTR=START;                                   3  198 of RETRIEVE
                  DO WHILE(¬((DIRECTORY_PTR=NULL)|(KEY_NAME=KEY)));   4  199
                     IF KEY_NAME>KEY THEN                             5  200
                     DIRECTORY_PTR=DOWN_PTR;                          5  201
                     ELSE                                             5  202
                     DIRECTORY_PTR=UP_PTR;                            5  202
                  END;                                                4  203
               END CHECKDIRET;                                        3  204
```

```
                                                                        3  204
0SCANST:        PROCEDURE(ST_PTR)RETURNS(POINTER);                      3  205
 /* SAME AS FOR STORAGE */                                              3  206
                DCL IND FIXED BINARY;                                   3  206
                DCL (ST_PTR,P) POINTER;                                 3  207
                STORAGE_PTR=ST_PTR;                                     3  208
                IND=1;                                                  3  209
                   DO WHILE(NAME(IND)¬=KEY);                            4  210
                   IND=IND+1;                                           4  211
                   END;                                                 4  212
                P=NEXT_PTR(IND);                                        3  213
                RETURN(P);                                              3  214
                END SCANST;                                             3  215
             END RETRSE;                                                2  216
1MERGE:      PROCEDURE;                                                 2  217
             DCL (LL,NN) FIXED BINARY;                                  2  218
                DO LL=1 TO M;                                           3  219
                   IF RETPTRS(LL)¬=NULL THEN                            4  220
                      DO;                                               5  221
                      NN=1;                                             5  222
                         DO WHILE((NN<=MM)&(WORKARRAY(NN)¬=RETPTRS(LL)));  6  223
                         NN=NN+1;                                       6  224
                         END;                                           6  225
                         IF NN>MM THEN                                  6  226
                            DO;                                         7  227
                            WORKARRAY(NN)=RETPTRS(LL);                  7  228
                            MM=NN;                                      7  229
                            END;                                        7  230
                      END;                                              5  231
                END;                                                    3  232
             END MERGE;                                                 2  233
1CHECK_FOR_KEY:                                                         2  234
             PROCEDURE(TITLE,UNARY);                                    2  234


             DCL TITLE CHAR(LENGTH_KEY_NAME);
             DCL UNARY BIT(1);                                          2  236
             DCL (LL,NN) FIXED BINARY;                                  2  237
                DO LL=1 TO M;                                           3  238
                   IF RETPTRS(LL)¬=NULL THEN                            4  239
                      DO;                                               5  240
                      STORAGE_PTR=RETPTRS(LL);                          5  241
                      NN=1;                                             5  242
                         DO WHILE((NN<=#KEYS)&(NAME(NN)¬=TITLE));       6  243
                         NN=NN+1;                                       6  244
                         END;                                           6  245
                         IF((NN<=#KEYS)&(UNARY='1'B))|((NN>#KEYS)&(UNARY=  6  246
 '0'B))THEN                                                             6  246
                         RETPTRS(LL)=NULL;                              6  247
                      END;                                              5  248
                END;                                                    3  249
             END CHECK_FOR_KEY;                                         2  250
          END RETREVE;                                                  1  251
```

Step 12

of RETRIEVE

```
SERIAL#:PROC ($INCR,$WHICH) RETURNS(PIC'9999999');
DCL   $INCR  FIXED DEC(7);
DCL $COUNTER(20) FIXED(7)STATIC INIT((20)0);
DCL $WHICH FIXED DEC(3);
$COUNTER($WHICH)=$COUNTER($WHICH) + $INCR;
RETURN($COUNTER($WHICH));
END;
```

```
*PROCESS('MACRO,NST,EXTREF,N=STORE');
 STORE:  PROCEDURE(STRING,DATA_PTR);
 /* THIS PROCEDURE TAKES A STRING OF CONCATENATED KEY NAMES IN "STRING"   1    2
 AND A POINTER TO AUXILLIARY DATA ("DATA_PTR") AND STORES THE TWO IN
 A MEMORY SPACE, EACH ENTRY OF WHICH IS CALLED A "STORAGE_ENTRY" .
 FURTHERMORE, EACH KEY NAME IN "STRING" IS ENTERED IN A                  1    2
 "DIRECTORY_ENTRY", THE DIRECTORY HAVING A "BRANCH AND BOUND"            1    2
 STRUCTURE. */                                                           1    2
         DCL STRING CHAR(*);                                             1    4
         DCL DATA_PTR POINTER;                                           1    5
     %INCLUDE INCLIB(DSEDIR);
         DCL (I,J) FIXED BINARY;                                         1    8
         DCL (P,LAST_ST_ENTRY_PTR) POINTER;                              1    9
         DCL CURRENT_NAME CHAR(LENGTH_KEY_NAME);                         1   10
         DCL IND FIXED BINARY;                                           1   11
         DCL #DIREN FIXED BIN EXT;
         DCL START POINTER EXTERNAL;                                     1   13
         DCL SCANST ENTRY(CHAR(LENGTH_KEY_NAME),POINTER) RETURNS (POINTE 1   14
 R);                                                                     1   14
         DCL CHECK_DIR_ST ENTRY(CHAR(LENGTH_KEY_NAME)) RETURNS (POINTER) 1   15
 ;                                                                       1   15
         DCL GENERATE_ENTRY ENTRY(CHAR(LENGTH_KEY_NAME)) RETURNS (POINTE 1   16
 R);                                                                     1   16
         DCL NULL BUILTIN;                                               1   17
         I,J=1;                                                          1   18     1
         N=LENGTH(STRING)/LENGTH_KEY_NAME;                               1   19
         ALLOCATE STORAGE_ENTRY;                                         1   20     2
         LAST_ST_ENTRY_PTR=STORAGE_PTR;                                  1   21
         DATA_PT=DATA_PTR;                                               1   22     3
            DO WHILE(I<=#KEYS);                                          2   25     4
           CURRENT_NAME=SUBSTR(STRING,J,LENGTH_KEY_NAME);                2   26
            STORAGE_ENTRY.NAME(I)=CURRENT_NAME;                          2   27
            STORAGE_ENTRY.NEXT_PTR=NULL;                                 2   28
               IF START=NULL THEN                                        3   29
                  DO;                                                    4   30
                  START=GENERATE_ENTRY(CURRENT_NAME);                    4   31
                  CALL ESTABLISHLINKS(CURRENT_NAME,START);               4   32
                  END;                                                   4   33
               ELSE                                                      3   34
                  DO;                                                    4   34
                  P=CHECK_DIR_ST(CURRENT_NAME);                          4   35   5 - 7
 /* P IS NOW THE POINTER TO THE DIRECTORY ENTRY OF THE CURRENT           4   36
 KEY NAME ("CURRENT_NAME")*/                                             4   36   8 - 11
                  CALL ESTABLISHLINKS(CURRENT_NAME,P);                   4   36
                  END;                                                   4   37
            I=I+1;                                                       2   38
            J=J+LENGTH_KEY_NAME;                                         2   39
            END;                                                         2   40
```

```
CHECK_DIR_ST:                                                          2   41
          PROCEDURE(KEY)RETURNS(POINTER);                              2   41
/* CHECK_DIR_ST GETS THE KEY AND STARTS SEARCHING THE DIRECTORY.       2   42
IF THE NAME IS IN THE DIRECTORY, THEN IT RETURNS THE POINTER TO        2   42
THE DIRECTORY ENTRY. IF NOT, IT GENERATES A NEW ENTRY IN THE PROPER    2   42
PLACE AND ESTABLISHES THE LINKS IN THE DIRECTORY, RETURNING THE        2   42
POINTER TO THE NEWLY CREATED ENTRY. */                                 2   42
          DCL KEY CHAR(LENGTH_KEY_NAME);                               2   42
          DCL (NAME_PTR,C) POINTER;                                    2   43
          DIRECTORY_PTR=START;                                         2   44     1
TEST:          IF KEY>KEY_NAME THEN                                    3   45     4
                  DO;                                                  4   46
                     IF UP_PTR=NULL THEN                               5   47     7
                        DO;                                            6   48


                        C=DIRECTORY_PTR;                               6   49
                        NAME_PTR=GENERATE_ENTRY(KEY);                  6   50     9
                        DIRECTORY_PTR=C;                               6   51
/*BRANCH AND BOUND METHOD IS USED FOR THE DIRECTORY */                 6   52
                        DIRECTORY_PTR->DIRECTORY_ENTRY.UP_PTR=NAME_PTR; 6  52
                        END;                                           6   53
                     ELSE                                              5   54
                        DO;                                            6   54     8
/* UP_PTR¬=NULL*/       DIRECTORY_PTR=UP_PTR;                          6   55
                        GO TO TEST;                                    6   56
                        END;                                           6   57
                  END;                                                 4   58
               ELSE                                                    3   59
                  IF KEY<KEY_NAME THEN                                 4   59     3
                     DO;                                               5   60
                        IF DOWN_PTR=NULL THEN                          6   61     5
                           DO;                                         7   62
                           C=DIRECTORY_PTR;                            7   63
                           NAME_PTR=GENERATE_ENTRY(KEY);               7   64     9
                           DIRECTORY_PTR=C;                            7   65
                           DIRECTORY_PTR->DIRECTORY_ENTRY.DOWN_PTR=NAME_ 7 66
PTR;                                                                   7   66
                           END;                                        7   67
                        ELSE                                           6   68
                           DO;                                         7   68     6
                           DIRECTORY_PTR=DOWN_PTR;                     7   69
                           GO TO TEST;                                 7   70
                           END;                                        7   71
                     END;                                              5   72
                  ELSE                                                 4   73     2
                     DO;                                               5   73
/* KEY=KEY_NAME : I.E. DIRECTORY ENTRY FOUND */                        5   74
                     NAME_PTR=DIRECTORY_PTR;                           5   74
                     END;                                              5   75
          RETURN(NAME_PTR);                                            2   76
          END CHECK_DIR_ST;                                            2   77
```

```
GENERATE_ENTRY:                                                        2    78
           PROCEDURE(TITLE)RETURNS(POINTER);                          2    78
/*THIS PROCEDURE ALLOCATES A NEW "DIRECTORY_ENTRY" FOR THE KEY NAME IN  2    79
"TITLE",INITIALIZES THE POINTERS IN IT TO NULL AND INCREMENTS THE      2    79
COUNT OF NUMBER OF DIRECTORY ENTRIES ("#DIREN")*/                      2    79
           DCL TITLE CHAR(LENGTH_KEY_NAME);                            2    79
           ALLOCATE DIRECTORY_ENTRY;                                   2    80   1
           KEY_NAME=TITLE;                                             2    81   2
           UP_PTR,DOWN_PTR,FIRST_IN_LIST=NULL;                         2    82   3
           #DIREN=#DIREN+1;                                            2    83
           RETURN(DIRECTORY_PTR);                                      2    94   4
           END GENERATE_ENTRY;                                         2    85
ESTABLISHLINKS:                                                        2    96
           PROCEDURE(KEY,DIR_PTR);                                     2    86
/*ESTABLISHLINKS PROCEDURE ADDS THE NEWLY CREATED "STORAGE_ENTRY"      2    87
WITH THE KEY NAME CURRENTLY IN "KEY" TO THE LINKED LIST IN WHICH THE   2    87
SAME KEY IS MENTIONED. THE FIRST STORAGE ENTRY IN THE LIST WITH THE    2    87
CURRENT KEY NAME IS POINTED TO BY THE CURRENT KEY NAME'S DIRECTORY     2    87
ENTRY.                                                                 2    87
THE POINTER TO THE DIRECTORY ENTRY IS "DIR_PTR"*/                      2    87
           DCL KEY CHAR(LENGTH_KEY_NAME);                              2    87
           DCL (DIR_PTR,ARROW) POINTER;                                2    88
           DIRECTORY_PTR=DIR_PTR;                                      2    89
              IF FIRST_IN_LIST=NULL THEN                               3    90
              FIRST_IN_LIST=LAST_ST_ENTRY_PTR;                         3    91
              ELSE                                                     3    92
                 DO;                                                   4    92


                 STORAGE_PTR=FIRST_IN_LIST;                            4    93
                 ARROW=SCANST(KEY,STORAGE_PTR);                        4    94
                    DO WHILE(ARROW¬=NULL);                             5    95
                    STORAGE_PTR=ARROW;                                 5    96
                    ARROW=SCANST(KEY,STORAGE_PTR);                     5    97
/* STORAGE_PTR TRAILS ARROW */                                         5    98
                    END;                                               5    98.
                 NEXT_PTR(IND)=LAST_ST_ENTRY_PTR;                      4    99
                 END;                                                  4   100
           STORAGE_PTR=LAST_ST_ENTRY_PTR;                              2   101
/* RESET "STORAGE_PTR" TO "LAST_STORAGE_ENTRY" SINCE IT WAS MODIFIED   2   102
BY SCANST */                                                           2   102
           END ESTABLISHLINKS;                                         2   102
SCANST:    PROCEDURE(TITLE,ST_PTR)RETURNS(POINTER);                    2   103
/*SCANST SCANS A STORAGE ENTRY UNTIL FINDS THE KEY GIVEN BY 'TITLE'*/  2   104
/*IT RETURNS THE VALUE OF THE POINTER ASSOCIATED WITH THAT KEY */      2   104
           DCL TITLE CHAR(LENGTH_KEY_NAME);                            2   104
           DCL ST_PTR POINTER;                                         2   105
           STORAGE_PTR=ST_PTR;                                         2   106
           IND=1;                                                      2   107
              DO WHILE((IND<=#KEYS)&(NAME(IND)¬=TITLE));               3   108
              IND=IND+1;                                               3   109
              END;                                                     3   110
           P=NEXT_PTR(IND);                                            2   111
           RETURN(P);                                                  2   112
           END SCANST;                                                 2   113
        END STORE;                                                     1   114
```

```
SUM: PRCC(X,I,L,H) RETURNS(PIC'SS99999999S');
DCL X(*) PIC'9999999';
DCL (I,L,H) FIXED BIN;
DCL TEMP     FIXED DEC(7) STATIC INIT(0);
DCL FIRSTSUM BIT(1) STATIC INIT('1'B);
IF FIRSTSUM THEN
DC;
  FIRSTSUM='0'B;
  TEMP=0;
END;
TEMP=TEMP+X(I);
IF I=H THEN FIRSTSUM='1'B;
RETURN(TEMP);
END;
```

```
TODAY: PROC  RETURNS(CHAR(13)VAR);
DCL  DATE BUILTIN;
DCL MONTH (12) CHAR(3) INIT('JAN','FEB','MAR','APR','MAY','JUN',
  'JUL','AUG','SEP','OCT','NOV','DEC');
DCL TDATE CHAR(6);
TDATE=CATE;
RETURN(MCNTH(SUBSTR(TCATE,3,2))||' '||SUBSTR(TDATE,5,2)||', '||
  SUBSTR(TDATE,1,2)  );
END;
```

Algorithm UNPACK

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=UNPACK');
 UNPACK:  PROC(NEXT_INDEX) RECURSIVE;
 /* EXTERNAL TO GENERATE_UNPACKING.                                       */
 DCL NEXT_INDEX    BIN FIXED(15); /* THIS PARAMETER TELLS US WHICH        */
                                  /* WHICH IS THE NEXT_INDEX TO USE       */
                                  /* IN A DO-LOOP OR FOR SUBSCRIPTING.*/
 %DCL LEN_DICT_ENTRY FIXED; %LEN_DICT_ENTRY=32;
 %DCL MAX_L_NAME FIXED; %MAX_L_NAME=10;
 %DCL LENGTH_KEY_NAME FIXED; %LENGTH_KEY_NAME=10;
 %INCLUDE INCLIBB(FTP);
 DCL FTP_PTR POINTER EXT;
 DCL NSTR          BIN FIXED(15) EXT;


 DCL PTR POINTER;
 DCL UNPKFLD ENTRY(BIN FIXED(15));
 DCL UNPKGRP ENTRY(BIN FIXED(15));
 DCL SYSERR ENTRY(CHAR(*));
     PTR=FTP_PTR;
     NSTR=NSTR+1;                                                          1
          IF TYPE(NSTR)='F' THEN CALL UNPKFLD(NEXT_INDEX);                 2
     ELSE IF TYPE(NSTR)='G' THEN CALL UNPKGRP(NEXT_INDEX);
     ELSE /* TYPE(NSTR) ISN''T 'F' OR 'G'.                               */
          CALL SYSERR('GENIOCP-UNPACK:  ILLEGAL TYPE ''' ||
                      TYPE(NSTR) ||
                      ''' FOR SUBSTRUCTURE ''' ||
                      NAME(NSTR) ||
                      '''');
 RETURN;
 END; /* UNPACK */
```

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=UNPKFLD');
 UNPKFLD:  PROC(NEXT_INDEX);
 /* EXTERNAL TO UNPACK.                                                */
 /* THIS PROCEDURE GENERATES CODE FOR UNPACKING FIELDS.                */
 /* THERE ARE 3 CASES OF INTEREST:                                     */
 /*             #SUBSCRIPTS(NSTR) = 0                                  */
 /*             #SUBSCRIPTS(NSTR) = 1                                  */
 /*             #SUBSCRIPTS(NSTR) = 2                                  */
 DCL NEXT_INDEX      BIN FIXED(15); /* THIS PARAMETER TELLS US WHICH   */
                                    /* IS THE NEXT_INDEX TO USE IN A   */
                                    /* DO-LOOP OR FOR SUBSCRIPTING.    */
 DCL #RECNAME CHAR(32) VAR EXT;
 DCL #RECSTRING CHAR(34) VAR EXT;
 DCL PLISTR CHAR(320) VAR;
 DCL SUBSCRIPT_STRING ENTRY RETURNS(CHAR(100) VAR);
 DCL NSTR BIN FIXED(15) EXT;
 DCL PTR POINTER;
 DCL SYSERR ENTRY(CHAR(*));
 DCL FTF_PTR POINTER EXT;
 %DCL LEN_DICT_ENTRY FIXED; %LEN_DICT_ENTRY=32;
 %DCL MAX_L_NAME FIXED; %MAX_L_NAME=10;
 %DCL LENGTH_KEY_NAME FIXED; %LENGTH_KEY_NAME=10;
 %INCLUDE INCLIBR(FTF);
 DCL PUSHSTK ENTRY(BIN FIXED(15));
 DCL POPSTK ENTRY;
 DCL WRPLI ENTRY(CHAR(*)VAR,CHAR(*));
 DCL INDXGEN ENTRY(BIN FIXED(15)) RETURNS(CHAR(3));
 DCL PICTURE ENTRY(BIN FIXED(15)) RETURNS(CHAR(10)VAR);
 DCL CHPTRLI ENTRY(CHAR(*)) RETURNS(CHAR(32) VAR);
 DCL BYTE_CALC ENTRY(BIN FIXED(15),CHAR(1),BIN FIXED(15));
      PTR=FTF_PTR;
 -             IF #SUBSCRIPTS(NSTR) = 0                                    3
               THEN DO;                                                    4
                    CALL GENERATE_MOVE_INSTRS;
                    RETURN;
                    END;
 -        ELSE IF #SUBSCRIPTS(NSTR) = 1                                    5
               THEN DO;
                    CALL PUSHSTK(NEXT_INDEX);
                    PLISTR='DO ' ||
                          INDXGEN(NEXT_INDEX) ||
                          '=1 TO ' ||
                          PICTURE(SUBSCRIPT1(NSTR)) ||
                          ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    CALL GENERATE_MOVE_INSTRS;
                    PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIEX');
               CALL POPSTK;
               RETURN;
               END;
 -     ELSE IF (#SUBSCRIPTS(NSTR)=2) & (SUBSCRIPT2(NSTR)=1)                  6
            THEN DO;
               PLISTR='CALL ' ||
                      CHPTRLB(EXIST_PROC(NSTR)) ||
                      ';';
               CALL WRPLI(PLISTR,'PLIEX');
               PLISTR='DO ' ||
                      INDXGEN(NEXT_INDEX) ||
                      '=1 TO EXIST.' ||
                      CHPTRLB(NAME(NSTR)) ||
                      ';';
               CALL WRPLI(PLISTR,'PLIEX');
               CALL GENERATE_MOVE_INSTRS;
               PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIEX');
               RETURN;
               END;
 -     ELSE IF #SUBSCRIPTS(NSTR) = 2                                         7
            THEN DO;
               CALL PUSHSTK(NEXT_INDEX);
               PLISTR='CALL ' ||
                      CHPTRLB(EXIST_PROC(NSTR)) ||
                      ';';
               CALL WRPLI(PLISTR,'PLIEX');
               PLISTR='DO ' ||
                      INDXGEN(NEXT_INDEX) ||
                      '=1 TO EXIST.' ||
                      CHPTRLB(NAME(NSTR)) ||
                      ';';
               CALL WRPLI(PLISTR,'PLIEX');
               CALL GENERATE_MOVE_INSTRS;
               PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIEX');
               CALL POPSTK;
               RETURN;
               END;
 -     ELSE /* #SUBSCRIPTS(NSTR) ISN'T 0 OR 1 OR 2.                      */
            CALL SYSERR('GENIOCD - UNPKFLD:  ILLEGAL #SUBSCRIPTS:  ' ||
                      PICTURE(#SUBSCRIPTS(NSTR)) ||
                      ' FOR SUBSTRUCTURE ''' ||
                      NAME(NSTR) ||
                      '''');
```

Algorithm

GEN-MOVE-INSTR-FOR-UNPACKING

```
1GENERATE_MOVE_INSTRS:  PROC;
  /* INTERNAL TO UNPACK_FIELD.                                    */
  /* THIS PROCEDURE GENERATES INSTRUCTIONS TO:                    */
  /*           TRANSFER INFORMATION;                              */
  /*           ADVANCE THE BUFFER POINTER (I).                    */
  DCL #BYTES          BIN FIXED(15);                              1
          IF DATA_TYPE(NSTR)='C'
          THEN IF FIELD_LEN_TYPE(NSTR)='F' /* FIXED LENGTH */     5
               THEN DO;
                    PLISTR=#RECNAME ||
                           '.' ||
                           CHPTRLB(NAME(NSTR)) ||
                           SUBSCRIPT_STRING ||
                           '=SUBSTR(' ||
                           #RECSTRING ||
                           ',I,' ||
                           PICTURE(MIN_LENGTH(NSTR)) ||


                           ');';
                    CALL WRPLI(PLISTR,'PLIEX');
                    PLISTR='I=I+' ||
                           PICTURE(MIN_LENGTH(NSTR)) ||
                           ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;
0              ELSE DO; /* CHARACTER - VARIABLE LENGTH */         2, 6
                    PLISTR='CALL ' ||
                           CHPTRLB(LEN_PROC(NSTR)) ||
                           ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    PLISTR=#RECNAME ||
                           '.' ||
                           CHPTRLB(NAME(NSTR)) ||
                           SUBSCRIPT_STRING ||
                           '=SUBSTR(' ||
                           #RECSTRING ||
                           ',I,LEN.' ||
                           CHPTRLB(NAME(NSTR)) ||
                           ');';
                    CALL WRPLI(PLISTR,'PLIEX');
                    PLISTR='I=I+LEN.' ||
                           CHPTRLB(NAME(NSTR)) ||
                           ';';
                    CALL WRPLI(PLISTR,'PLIEX');
                    RETURN;
                    END;
```

AD-A032 638 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA PA F/G 9/2
AUTOMATIC GENERATION OF BUSINESS DATA-PROCESSING PROGRAMS FROM --ETC(U)
OCT 76 N A RIN N00014-76-C-0416

UNCLASSIFIED 76-05 NL

6 OF 6
AD
A032638

END
DATE
FILMED
1-77

```
-      ELSE IF DATA_TYPE(NSTR)='B' /* BINARY */                                    3
           THEN DO;                                                                7
                CALL BYTE_CALC(MIN_LENGTH(NSTR),'B',#BYTES);
                PLISTR='UNSPEC(' ||
                        #RECNAME ||
                        '.' ||
                        CHRTRLB(NAME(NSTR)) ||
                        SUBSCRIPT_STRING ||
                        ')=UNSPEC(SUBSTR(' ||
                        #RECSTRING ||
                        ',I,' ||
                        PICTURE(#BYTES) ||
                        '));';
                CALL WRPLI(PLISTR,'PLIEX');
                PLISTR='I=I+' ||
                        PICTURE(#BYTES) ||
                        ';';
                RETURN;
                END;
-      ELSE IF DATA_TYPE(NSTR)='N' /* NUMERIC */                                   1,.5
           THEN DO;
                PLISTR=#RECNAME ||
                        '.' ||
                        CHRTRLB(NAME(NSTR)) ||
                        SUBSCRIPT_STRING ||
                        '=SUBSTR(' ||
                        #RECSTRING ||
                        ',I,' ||
                        PICTURE(MIN_LENGTH(NSTR)) ||
                        ');';
                CALL WRPLI(PLISTR,'PLIEX');
                PLISTR='I=I+' ||
                        PICTURE(MIN_LENGTH(NSTR)) ||
                        ';';


                CALL WRPLI(PLISTR,'PLIEX');
                RETURN;
                END;
-      ELSE IF DATA_TYPE(NSTR)='F' /* FIXED-DECIMAL */                                 4
           THEN DO;
                CALL BYTE_CALC(MIN_LENGTH(NSTR),'B',#BYTES);                           .8
                PLISTR='UNSPEC(' ||
                        #RECNAME ||
                        '.' ||
                        CHRTRLB(NAME(NSTR)) ||
                        SUBSCRIPT_STRING ||
                        ')=UNSPEC(SUBSTR(' ||
                        #RECSTRING ||
                        ',I,' ||
                        PICTURE(#BYTES) ||
                        '));';
                CALL WRPLI(PLISTR,'PLIEX');
                PLISTR='I=I+' ||
                        PICTURE(#BYTES) ||
                        ';';
                RETURN;
                END;
       ELSE /* DATA_TYPE(NSTR) ISN'T 'C' OR 'B' OR 'F'. */
           CALL SYSERR('GENIDCD:  ILLEGAL DATA_TYPE ''' ||
                        DATA_TYPE(NSTR) ||
                        ''' FOR SUBSTRUCTURE ''' ||
                        NAME(NSTR) ||
                        '''');
   END; /* GENERATE_MOVE_INSTRS */
   END; /* UNPACK_FIELD */
```

Algorithm

UNPACKGRP

```
*PROCESS('MACRO,EXTREF,SM=(2,72,1),N=UNPKGRP');
 UNPKGRP:  PROC(NEXT_INDEX) RECURSIVE;
 %DCL LEN_DICT_ENTRY FIXED; %LEN_DICT_ENTRY=32;
 %DCL MAX_L_NAME FIXED; %MAX_L_NAME=10;
 %DCL LENGTH_KEY_NAME FIXED; %LENGTH_KEY_NAME=10;
 /* EXTERNAL TO UNPACK.                                                */
 /* THIS PROCEDURE GENERATES THE CODE REQUIRED FOR A GROUP, AND        */
 /* CALLS UNPACK TO GENERATE CODE FOR ITS MEMBERS.                     */
 /* THERE ARE 3 CASES OF INTEREST:                                     */
 /*                   #SUBSCRIPTS(NSTR) = 0                            */
 /*                   #SUBSCRIPTS(NSTR) = 1                            */
 /*                   #SUBSCRIPTS(NSTR) = 2                            */
 DCL NEXT_INDEX BIN FIXED(15);
 DCL NSTR BIN FIXED(15) EXT;
 DCL I             BIN FIXED(15);
 DCL LOCAL_ARITY   BIN FIXED(15);
 DCL PLISTR CHAR(320) VAR;
 DCL UNPACK ENTRY(BIN FIXED(15));
 DCL INDXGEN ENTRY(BIN FIXED(15)) RETURNS(CHAR(3));
 DCL PICTURE ENTRY(BIN FIXED(15)) RETURNS(CHAR(10) VAR);
 DCL WRPLI ENTRY(CHAR(*)VAR,CHAR(*));
 DCL PUSHSTK ENTRY(BIN FIXED(15));
 DCL POPSTK ENTRY;
 DCL CHRTRLR ENTRY(CHAR(*)) RETURNS(CHAR(32)VAR);
 DCL SYSERR ENTRY(CHAR(*));
 DCL FTR_PTR POINTER EXT;
 %INCLUDE INCLIBR(FTR);
      PTR=FTR_PTR;
          IF #SUBSCRIPTS(NSTR) = 0                                           10
          THEN DO;
               LOCAL_ARITY=ARITY(NSTR);
               DO I=1 TO LOCAL_ARITY;
                   CALL UNPACK(NEXT_INDEX);


               END;
               RETURN;
               END;
      ELSE IF #SUBSCRIPTS(NSTR) = 1
          THEN DO;                                                           11-2
               CALL PUSHSTK(NEXT_INDEX);
               PLISTR='DO ' ||
                     INDXGEN(NEXT_INDEX) ||
                     '=1 TO ' ||
                     PICTURE(SUBSCRIPT2(NSTR)) ||
                     ';';
               CALL WRPLI(PLISTR,'PLIFX');
               LOCAL_ARITY=ARITY(NSTR);
               DO I=1 TO LOCAL_ARITY;
                   CALL UNPACK(NEXT_INDEX+1);
               END;
               PLISTR='END;';
               CALL WRPLI(PLISTR,'PLIFX');
               CALL POPSTK;
               RETURN;
               END;
```

```
  -     ELSE IF (#SUBSCRIPTS(NSTR)=2) & (SUBSCRIPT2(NSTR)=1)
              THEN DO:
                    PLISTR='CALL ' ||                                          11-3
                          CHPTRLB(EXIST_PROC(NSTR)) ||
                          ';';
                    CALL WRPL1(PLISTR,'PL1EX');
                    PLISTR='DO ' ||
                          INDXGEN(NEXT_INDEX) ||
                          '=1 TO EXIST.' ||
                          CHPTRLB(NAME(NSTR)) ||
                          ';';
                    CALL WRPL1(PLISTR,'PL1EX');
                    LOCAL_ARITY=ARITY(NSTR);
                    DO I=1 TO LOCAL_ARITY;
                         CALL UNPACK(NEXT_INDEX+1);
                    END;
                    PLISTR='END;';
                    CALL WRPL1(PLISTR,'PL1EX');
                    RETURN;
                    END;
  -     ELSE IF #SUBSCRIPTS(NSTR) = 2                                         11- 4
              THEN DO:
                    CALL PUSHSTK(NEXT_INDEX);
                    PLISTR='CALL ' ||
                          CHPTRLB(EXIST_PROC(NSTR)) ||
                          ';';
                    CALL WRPL1(PLISTR,'PL1EX');
                    PLISTR='DO ' ||
                          INDXGEN(NEXT_INDEX) ||
                          '=1 TO EXIST.' ||
                          CHPTRLB(NAME(NSTR)) ||
                          ';';
                    CALL WRPL1(PLISTR,'PL1EX');
                    LOCAL_ARITY=ARITY(NSTR);
                    DO I=1 TO LOCAL_ARITY;
                         CALL UNPACK(NEXT_INDEX+1);
                    END;
                    PLISTR='END;';
                    CALL WRPL1(PLISTR,'PL1EX');
                    CALL POPSTK;
                    RETURN;
                    END;

  -     ELSE /* #SUBSCRIPTS(NSTR) ISN'T 0 OR 1 OR 2. */
              CALL SYSERR('GENIOCR - UNPKGRP:  ILLEGAL #SUBSCRIPTS:  ' ||
                          PICTURE(#SUBSCRIPTS(NSTR)) ||
                          ' FOR SUBSTRUCTURE ''' ||
                          NAME(NSTR) ||
                          '''');
     END; /* UNPACK_GROUP */
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=WRDCL');
WRDCL: PROC(LEVEL,STRING);
/*THIS PROCEDURE WRITES OUT THE DECLARATION IN 'STRING' INDENTING BY
'LEVEL', AND BREAKING UP'STRING' IF NECESSARY TO FIT ON
MULTIPLE CARDS*/
DCL STRING CHAR(*) VAR;
DCL LEVEL FIXED DEC;
 DCL          LEVEL_STRING CHAR(3);
/* LEVEL_STRING IS THE STRING EXPRESSION OF THE LEVEL */                       3   29
DCL BLANKS CHAR(71) STATIC INIT(' ');
DCL INDENT CHAR(71) VAR;
DCL 1 PLI_LINE STATIC,
      2 CC CHAR(1) INIT(' '),
      2 TEXT CHAR(71),
      2 DCL_IND CHAR(3) INIT('DCL'),
      2 DCL# PIC'99999' INIT(0);
   IF LEVEL>0 THEN
   DO;
      PUT STRING(LEVEL_STRING) EDIT(LEVEL)(F(3));
      STRING=LEVEL_STRING||' '||STRING;
   END;
DCL#=DCL# + 1;
    IF LEVEL=0 THEN INDENT='';
     ELSE INDENT=SUBSTR(BLANKS,1,LEVEL);
     L=LEVEL + LENGTH(STRING);
     DO WHILE (L>71);
        DO I=71 TO 1 BY -1
           WHILE(INDEX(' ()',SUBSTR(STRING,I,1))=0);
        END;
        IF I=0B THEN
        DO;
        /*CANT FIND BREAK POINT: CONTINUE COLUMN 2*/
           I=71; INDENT='';
           LEVEL=0;
        END;
         /*FOUND BREAK POINT AT I*/


           TEXT=INDENT||SUBSTR(STRING,1,I);
           WRITE FILE(PLIDCL) FROM(PLI_LINE);
           DCL#=DCL#+1;
           STRING=SUBSTR(STRING,I+1);
           L=LEVEL + LENGTH(STRING);
     END;
     TEXT=INDENT||STRING;
     WRITE FILE(PLIDCL) FROM(PLI_LINE);
     RETURN;
     END;
```

```
*PROCESS('NST,MACRO,SM=(2,72,1),N=WRPL1');
 WRPL1:  PROC(STRING,FILE);
-/* THIS ROUTINE IS RESPONSIBLE FOR WRITING GENERATED PL1 STATEMENTS   */
 /* TO VARIOUS TEMPORARY FILES.                                        */
-/* INPUTS ARE:                                                        */
 /* STRING - THE PL1 TARGET STRING;                                    */
 /* FILE - NAME OF THE DESIRED OUTPUT FILE                             */
-/* EACH CALL OF WRPL1 WRITES ONE OR MORE CARDS OF PL1 CODE.  IF       */
 /* STRING IS LONGER THAN 71 CHARACTERS, THEN TWO OR MORE CARDS ARE    */
 /* WRITTEN.  SUCCESSIVE CARDS ARE STRUNG TOGETHER, COLUMN 72 BEING    */
 /* CONSIDERED ADJACENT TO COLUMN 2 OF THE NEXT CARD.                  */
-/* EACH GENERATED CARD HAS 2 SEQUENCE FIELDS:                         */
 /*    CALL# - COLS 73-76 - TELLING ON WHICH CALL TO WRPL1 THIS CARD   */
 /*            WAS GENERATED; THIS NUMBER IS THE SAME FOR ALL CARDS    */
 /*            GENERATED IN THE SAME CALL OF WRPL1.                    */
 /*    CARD# - COLS 77-80 - TELLING THE ABSOLUTE ORDER IN WHICH THIS   */
 /*            CARD WAS CREATED.                                       */
-/* THE ENTRY DECLARATION FOR WRPL1 IS:                                */
 /*        DCL WRPL1 ENTRY(CHAR(*) VARYING,CHAR(*));                   */
-DCL STRING CHAR(*) VARYING;
 DCL FILE   CHAR(*);
 DCL TEMP CHAR(N) VARYING CTL;
 DCL (PL1EX,PL1ON,PL1PROC) FILE OUTPUT RECORD SEQL;
 DCL 1 OUTREC STATIC,
       2 PRINTER_CONTROL CHAR(1) INIT(' '),
       2 TEXT             CHAR(71),
       2 CALL#            PIC'9999' INIT(0),
       2 CARD#            PIC'9999' INIT(0);
-     CALL#=CALL#+1;
      N=LENGTH(STRING);
      ALLOCATE TEMP;
      TEMP=STRING;  /* THIS SIMULATES CALL-BY-VALUE */
-     DO WHILE(LENGTH(TEMP)>71);
         DO I=71 TO 1 BY -1 WHILE(INDEX(' ;,()=',SUBSTR(TEMP,I,1))=0);
         END;
         IF I=0 THEN  /* CANT FIND BREAKPOINT, CONTINUE COL. 2  */
            I=71;
         /* FOUND BREAKPOINT AT I  */
         TEXT=SUBSTR(TEMP,1,I);
         CALL WRT;
         TEMP=SUBSTR(TEMP,I+1);
      END;
      TEXT=TEMP;
      CALL WRT;
 FREE TEMP;
-WRT: PROC;
         CARD#=CARD#+1;
         IF FILE = 'PL1EX' THEN WRITE FILE(PL1EX) FROM(OUTREC);
         ELSE IF FILE = 'PL1ON' THEN WRITE FILE(PL1ON) FROM(OUTREC);
         ELSE IF FILE = 'PL1PROC' THEN WRITE FILE(PL1PROC)
                                                    FROM(OUTREC);
         ELSE /* INVALID FILE PARAMETER */
```